Windows PowerShell Networking Guide

>

PowerShell.org

Table of Contents

Created by Microsoft's "The Scripting Guy," Ed Wilson, this guide helps you understand how PowerShell can be used to manage the networking aspects of your server and client computers.

PowerShell Networking Guide

By Ed Wilson

Cover design by Nathan Vonnahme

Created by Microsoft's "The Scripting Guy," Ed Wilson, this guide helps you understand how PowerShell can be used to manage the networking aspects of your server and client computers.

Was this book helpful? The author(s) kindly ask(s) that you make a tax-deductible (in the US; check your laws if you live elsewhere) donation of any amount to The DevOps Collective to support their ongoing work.

Check for Updates! Our ebooks are often updated with new and corrected content. We make them available in three ways:

- Our main, authoritative GitHub organization, with a repo for each book. Visit https://github.com/devops-collective-inc/
- Our GitBook page, where you can browse books online, or download as PDF, EPUB, or MOBI. Using the online reader, you can link to specific chapters. Visit https://www.gitbook.com/@devopscollective
- On LeanPub, where you can download as PDF, EPUB, or MOBI (login required), and "purchase" the books to make a donation to DevOps Collective. You can also choose to be notified of updates. Visit https://leanpub.com/u/devopscollective

GitBook and LeanPub have slightly different PDF formatting output, so you can choose the one you prefer. LeanPub can also notify you when we push updates. Our main GitHub repo is authoritative; repositories on other sites are usually just mirrors used for the publishing process. GitBook will usually contain our latest version, including not-yet-finished bits; LeanPub always contains the most recent "public release" of any book.

Windows PowerShell Basics - Introduction

Windows PowerShell is not new technology. Windows PowerShell 4.0 ships in Windows 8.1 and in Windows Server 2012 R2, it has therefore been around for a while. Windows PowerShell is an essential admin tool designed specifically for Windows administration. By learning to use Windows PowerShell, network administrators quickly gain access to information from Windows Management Instrumentation, Active Directory and other essential sources of information. Additionally, Microsoft added Windows PowerShell support to the Common Criteria requirements for shipping enterprise applications. Therefore, to manage Microsoft Exchange, Azure, SQL Server, and others one needs to know and to understand how to use Windows PowerShell. In the networking world, this knowledge is also a requirement for managing DNS, DHCP, Network Adapters, and other components.

So what are the basics of Windows PowerShell that I need to know?

Windows PowerShell comes in two flavors - the first is an interactive console (sort of like a KORN or a BASH console in the UNIX world) built into the Windows command prompt. The Windows PowerShell console makes it simple to type short commands and to receive sorted, filtered, formatted results. These results easily display to the console, but can redirect to XML, CSV, or text files. The Windows PowerShell console offers several advantages such as speed, low memory overhead, and a comprehensive transcription service that records all commands and command output.

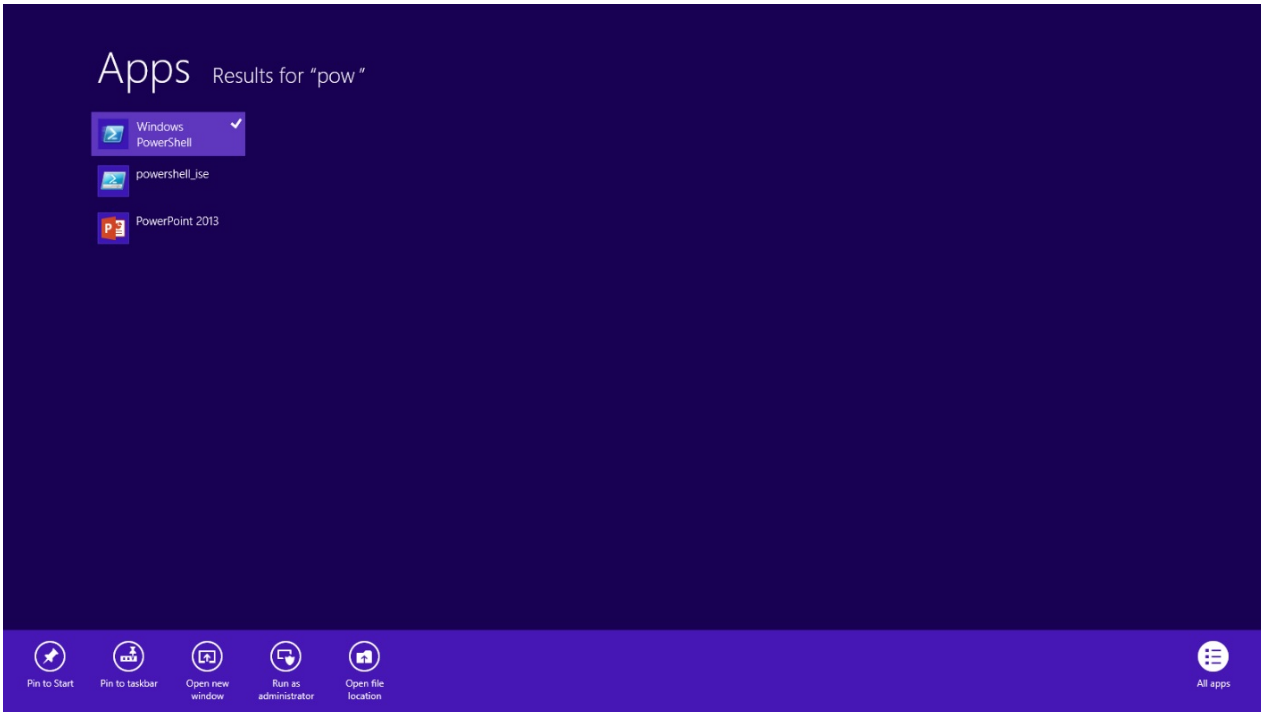
There is also the Windows PowerShell ISE. The Windows PowerShell ISE is an Integrated Scripting Environment, but this does not mean you must use it to write scripts. In fact, many Windows PowerShell users like to write their code in the Windows PowerShell ISE to take advantage of the color syntax-highlighting, drop down lists, and automatic parameter revelation features. In addition, the Windows PowerShell ISE has a feature, called the _Show Command Add-On _that permits using a mouse to create Windows PowerShell commands from a graphical environment. Once created, the command either runs directly, or adds to the script pane (the choice is up to you).

Working with Windows PowerShell

On Windows 8 or on Windows Server 2012 Windows PowerShell 3.0 already exists. On Windows 8.1 Windows PowerShell 4.0 is installed, as it is on Windows Server 2012 R2. Windows 8 (and 8.1) you only need to type the first few letters of the word PowerShell on the Start screen before Windows PowerShell appears as an option. The figure appearing here illustrates this point. I only typed _pow _before the Start screen search box changes to offer Windows PowerShell and an option.



Because navigating to the Start screen and typing _pow _each time I want to launch Windows PowerShell is a bit cumbersome, I prefer to Pin the Windows PowerShell console (and the Windows PowerShell ISE) to both the Start page and to the Windows desktop taskbar. This technique of pinning shortcuts to the applications provides single click access to Windows PowerShell from wherever I may be working.



On Windows Server 2012 (and on Windows Server 2012 R2), it is not necessary to go through the Start screen / Search routine because an icon for the Windows PowerShell console exists by default on the taskbar of the desktop.

NOTE : The Windows PowerShell ISE (the script editor) does not exist by default on Windows Server 2012 and Windows Server 2012 R2. You add the Windows PowerShell ISE as a feature.

Windows PowerShell Basics - Security issues with Windows PowerShell

There are two ways of launching Windows PowerShell - as an administrator and as a normal user. It is a best practice when starting Windows PowerShell to start it with minimum rights. On Windows 8 (and on Windows 7) this means simply clicking on the Windows PowerShell icon. It opens as a non-elevated user (even if you are logged on with Administrative rights). On Windows Server 2012, Windows PowerShell automatically launches with the rights of the current user and therefore if you are logged on as a Domain Administrator, the Windows PowerShell console launches with Domain Administrator rights.

Running as a normal (non-elevated) user

Because Windows PowerShell adheres to Windows security constraints, a user of Windows PowerShell cannot do anything that the user account does not have permission to do. Therefore, if you are a non-elevated normal user, you will not have rights to do things like install printer drivers, read from the Security log, or change system time.

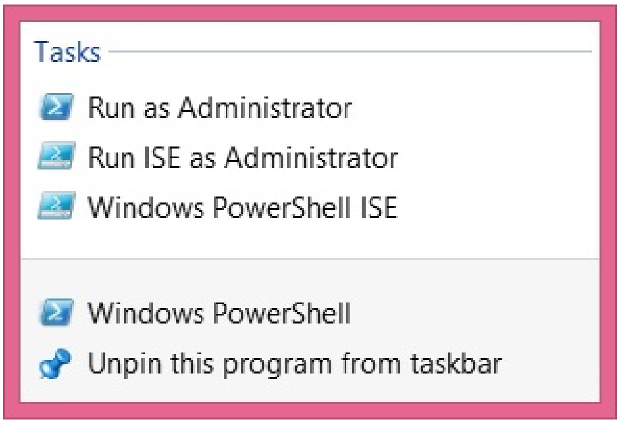
Even if you are an administrator on the local Windows 8 (or Windows 7) desktop machine and you do not launch Windows PowerShell with admin rights, you will get errors when attempting to do things like see the configuration of your disk drives. This command and associated error appears here.

```
PS C:\> get-disk
get-disk : Access to a CIM resource was not available to the client.
At line:1 char:1
+ get-disk
+ ~~~~~~~~
  + CategoryInfo     : PermissionDenied: (MSFT\_Disk:ROOT/Microsoft/Windows/S
  torage/MSFT\_Disk) [Get-Disk], CimException
  + FullyQualifiedErrorId : MI RESULT 2,Get-Disk
```

TIP : There is an inconsistency with errors arising when attempting to run cmdlets that require elevated rights. For example, when inside a non-elevated Windows PowerShell console, the error from Get-Disk is _Access to a CIM resource was not available to the client. _The error from Stop-Service is _Cannot open xxx service on computer. _While the Get-VM cmdlets simply returns no information (an no error). Therefore, as a first step in troubleshooting, check for console rights.

Launching PowerShell with Admin rights

When you need to perform tasks that require Admin rights, you need to start the Windows PowerShell console with admin rights. To do this, right click on the Windows PowerShell icon (from either the one pinned to the task bar, the start page, or even from the one found from the Start / Search page) and select the _Run As Administrator _option from the action menu. The great thing about this technique is that it permits launching either the Windows PowerShell console (the first item on the menu) as an Administrator, or from the same screen you can launch the Windows PowerShell ISE as an Administrator. This appears in the figure that follows.



Once you launch the Windows PowerShell console with admin rights, the User Account Control dialog box appears seeking permission to allow Windows PowerShell to make changes to the computer. In reality, Windows PowerShell is not making changes to the computer - not yet. But using Windows PowerShell you can certainly make changes to the computer - if you have the rights, and this is what the dialog is prompting you for.

NOTE : It is possible to avoid this prompt by turning off User Account Control (UAC). However, UAC is a significant security feature, and therefore I do not recommend disabling UAC. We have fine-tuned it in Windows 7 and continuing through Windows 8.1 and greatly reduced the number of UAC prompts (from the number that used to exist in the introduction of UAC on Windows Vista. This is not "your grandma's UAC".)

Now that you are running Windows PowerShell with admin rights, you can do anything your account has permission to do. Therefore, if you were to, for example, run the Get-Disk cmdlets, you would see information similar to the following appear.

```
PS C:\> get-disk


Number Friendly Name               OperationalS Total Size Partition
                        tatus          Style
------ -------------               ------------ ---------- -----------
0   INTEL SSDSA2BW160G3L           Online     149.05 GB MBR
```

Windows PowerShell Basics - Using PowerShell cmdlets

PowerShell cmdlets all work in a similar fashion. This simplifies their use. All Windows PowerShell cmdlets have a two-part name. The first part is a verb (not always strictly a grammatical verb however). The *verb* indicates the action for the command to take. Examples of verbs include Get, Set, Add, Remove, or Format. The *noun* is the thing to which the action will apply. Examples of nouns include Process, Service, Disk, or NetAdapter. A dash combines the verb with the noun to complete the Windows PowerShell command. Windows PowerShell commands, named cmdlets (pronounced command let), because they behave like small commands or programs are used standalone, or pieced together via a mechanism called the *pipeline* (refer to chapter two for the use of the *pipeline*).

The most common verb - `Get`

Out of nearly 2,000 cmdlets (and functions) on Windows 8, over 25 percent of them use the verb *Get.* The verb *Get* retrieves information. The Noun portion of the cmdlet specifies the information retrieved. To obtain information about the processes on your system, open the Windows PowerShell console by either clicking on the Windows PowerShell icon on the task bar (or typing PowerShell on the start screen of Windows 8 to bring up the search results for Windows PowerShell (as illustrated earlier)). Once the Windows PowerShell console appears, run the Get-Process cmdlet. To do this, use the Windows PowerShell Tab Completion feature to complete the cmdlet name. One the cmdlet name appears, press the key to cause the command to execute.

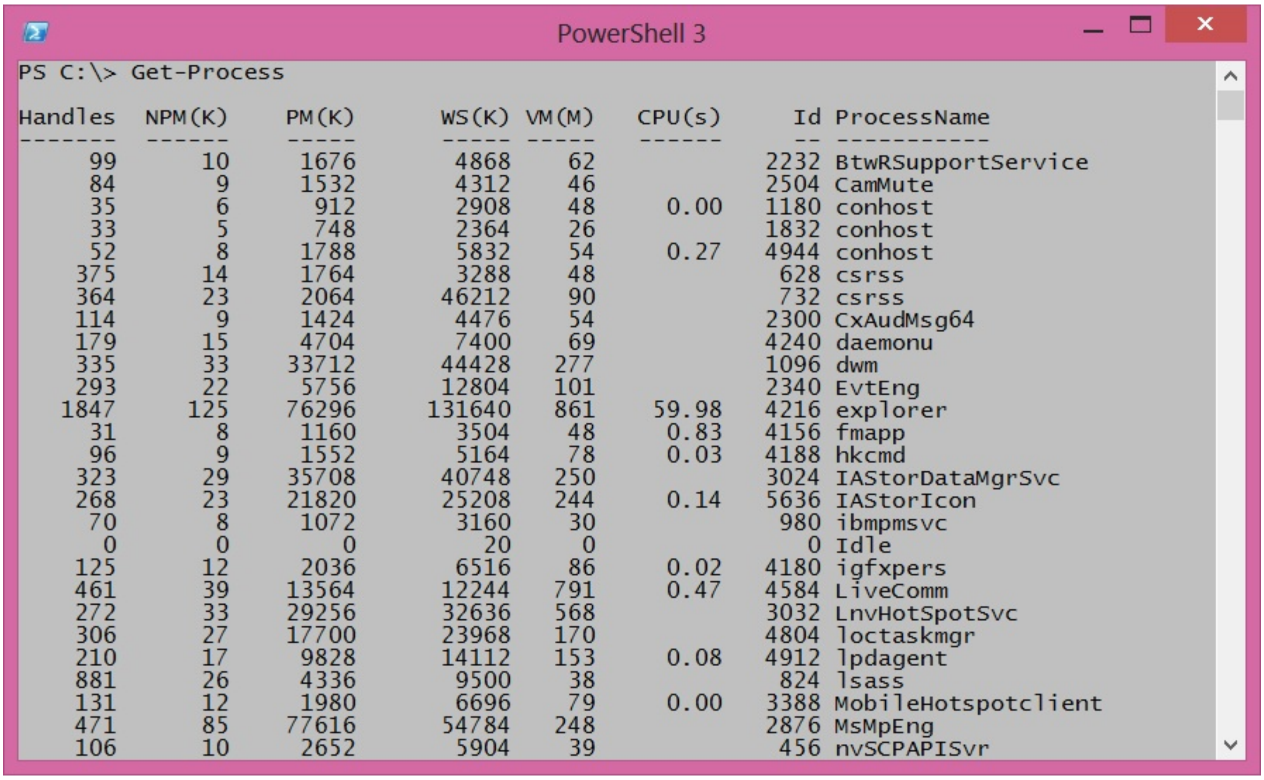
NOTE : The Windows PowerShell Tab Completion feature is a great time saver. It not only saves time (by reducing the need for typing) but it also helps to ensure accuracy, because Tab Completion accurately resolves cmdlet names - it is sort of like a spell checker for cmdlet names. For example, attempting to type a cmdlet name such as Get-NetAdapterEncapsulatedPacketTaskOffload accurately (for me anyway) could be an exercise in frustration. But using tab completion, I only have to type Get-Net and I hit the key about six times and the correctly spelled cmdlet name appears in the Windows PowerShell console. Learning how to quickly, and efficiently use the tab completion is one of the keys to success in using Windows PowerShell.

Finding process information

To use the Windows PowerShell Tab Completion feature to enter the Get-Process cmdlet name onto the Windows PowerShell console command line, type the following on the first line of the Windows PowerShell console:

```
Get-Pro + <tab> + <ENTER>
```

The Get-Process command and the associated output from the cmdlet appear in the figure that follows.



To find information about Windows services, use the verb **Get** and the **noun** service. To type the cmdlet name, type the following:

```
Get-Servi + <TAB> + <ENTER>
```

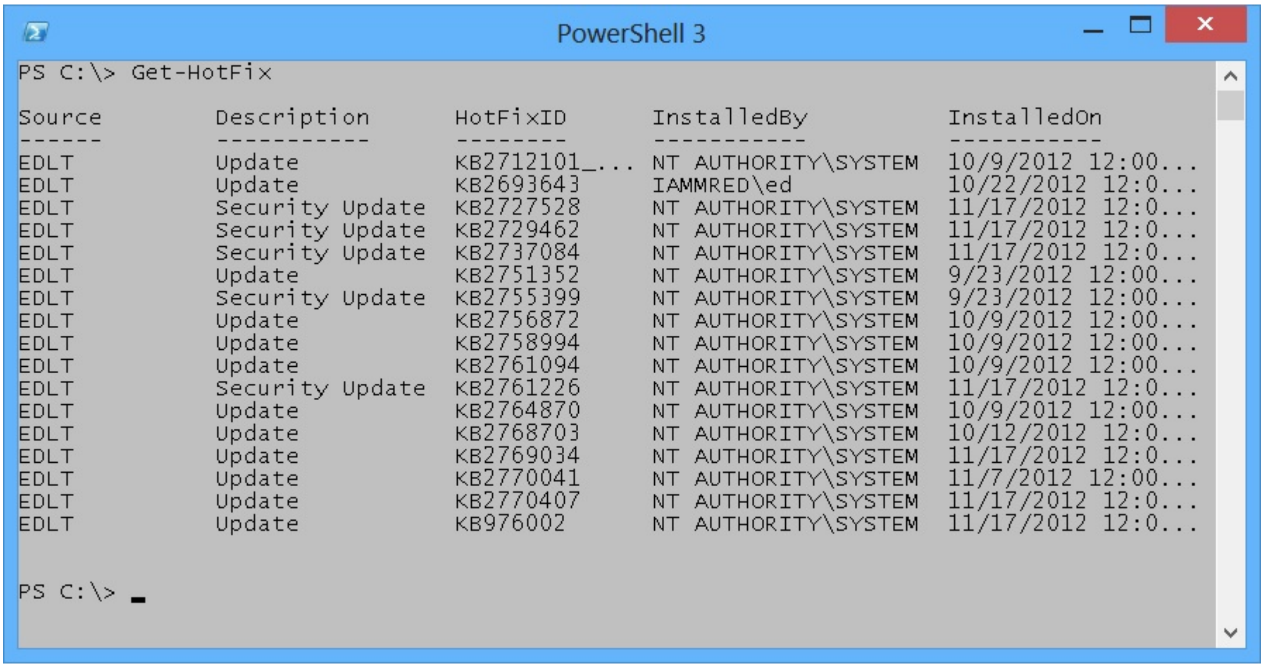
NOTE : It is a Windows PowerShell convention to use singular nouns. While not universally applied (my computer has around 50 plural nouns) it is a good place to start. So if you are not sure if a noun (or parameter) is singular or plural, choose the singular - most of the time you will be correct.

Identifying installed Windows Hotfixes

To find a listing of Windows Hotfixes applied to the current Windows installation, use the Get-Hotfix cmdlet (the **verb** is Get and the **noun** is Hotfix). Inside the Windows PowerShell console, type the following:

```
Get-Hotf + <TAB> + <ENTER>
```

The command, and the output associated with the command appear here.



```
PowerShell 3
PS C:\> Get-HotFix

Source        Description      HotFixID      InstalledBy          InstalledOn
------        -----------      --------      -----------          -----------
EDLT          Update           KB2712101_... NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Update           KB2693643     IAMMRED\ed           10/22/2012 12:0...
EDLT          Security Update  KB2727528     NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT          Security Update  KB2729462     NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT          Security Update  KB2737084     NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT          Update           KB2751352     NT AUTHORITY\SYSTEM  9/23/2012 12:00...
EDLT          Security Update  KB2755399     NT AUTHORITY\SYSTEM  9/23/2012 12:00...
EDLT          Update           KB2756872     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Update           KB2758994     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Update           KB2761094     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Security Update  KB2761226     NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT          Update           KB2764870     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Update           KB2768703     NT AUTHORITY\SYSTEM  10/12/2012 12:0...
EDLT          Update           KB2769034     NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT          Update           KB2770041     NT AUTHORITY\SYSTEM  11/7/2012 12:00...
EDLT          Update           KB2770407     NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT          Update           KB976002      NT AUTHORITY\SYSTEM  11/17/2012 12:0...

PS C:\> _
```

Get detailed service information

To find information about services on the system, use the Get-Service cmdlet. Once again, it is not necessary to type the entire command. The following command uses Tab Expansion to complete the Get-Service command and to execute it.

```
Get-Servi + <TAB> + <ENTER>
```

NOTE : The efficiency of Tab Expansion depends upon the number of cmdlets, functions, or modules installed on the computer. As more commands become available, the efficiency of Tab Expansion reduces correspondingly.

The following (truncated) output appears following the Get-Service cmdlet.

```
PS C:\> Get-Service


Status   Name          DisplayName
------   ----          -----------
Running AdobeActiveFile... Adobe Active File Monitor V6
Stopped AeLookupSvc     Application Experience
Stopped ALG         Application Layer Gateway Service
Stopped AllUserInstallA... Windows All-User Install Agent
<TRUNCATED OUTPUT>
```

Identifying installed network adapters

To find information about network adapters on your Windows 8 (or Windows Server 2012) machine, use the Get-NetAdapter cmdlet. Using Tab Expansion, type the following:

```
Get-NetA + <TAB> + <ENTER>
```

The command and associated output appear here.

```
PS C:\> Get-NetAdapter


Name           InterfaceDescription          ifIndex Status
----           --------------------          ------- ------
Network Bridge      Microsoft Network Adapter Multiplexo...   29 Up
Ethernet         Intel(R) 82579LM Gigabit Network Con...   13 Not Pre...
vEthernet (WirelessSwi... Hyper-V Virtual Ethernet Adapter #4     31 Up
vEthernet (External Sw... Hyper-V Virtual Ethernet Adapter #3     23 Not Pre...
vEthernet (InternalSwi... Hyper-V Virtual Ethernet Adapter #2     19 Up
Bluetooth Network Conn... Bluetooth Device (Personal Area Netw...   15 Disconn...
Wi-Fi           Intel(R) Centrino(R) Ultimate-N 6300...   12 Up
```

Retrieving detected network connection profiles

If you want to see the network connection profile that Windows 8 (or Windows Server 2012) detected for each interface, use the Get-NetConnectionProfile cmdlet. To run this command, use the following command with Tab Expansion.

```
Get-NetC + <TAB> + <ENTER>
```

The command and associated output appear here.

```
PS C:\> Get-NetConnectionProfile



Name             : Unidentified network
InterfaceAlias   : vEthernet (InternalSwitch)
InterfaceIndex   : 19
NetworkCategory  : Public
IPv4Connectivity : NoTraffic
IPv6Connectivity : NoTraffic


Name             : Network 10
InterfaceAlias   : vEthernet (WirelessSwitch)
InterfaceIndex   : 31
NetworkCategory  : Public
IPv4Connectivity : Internet
IPv6Connectivity : NoTraffic
```

NOTE : Windows PowerShell is not case sensitive. There are a few instances where case sensitivity is an issue (for example when using Regular Expressions) but cmdlet names, parameters and values are not case sensitive. Windows PowerShell convention uses a combination of upper case and lower case letters (generally at syllable breaks in long noun names such as NetConnectionProfile) but this is not a requirement for Windows PowerShell to interpret accurately the command. This combination of upper case and lowercase letters are for readability. If you use Tab Expansion, Windows PowerShell automatically converts the commands to this fashion.

Getting the current culture settings

There are two types of culture settings on a typical Windows computer. There are the culture settings for the current culture settings. This includes information about the keyboard layout, and the display format of items such as numbers, currency, and dates. To find the value of these cultural settings, use the Get-Culture cmdlet. To call the Get-Culture cmdlet using Tab Expansion to complete the command, type the following on the current line of the Windows PowerShell console:

```
Get-Cu + <TAB> + <ENTER>
```

When the command runs basic information such as the Language Code ID number (LCID), the name of the culture settings, as well as the display name of the culture settings return to the Windows PowerShell console. The command and associated output appears here.

```
PS C:\> Get-Culture


LCID        Name        DisplayName
----        ----        -----------
1033        en-US       English (United States)
```

The second culture related grouping of information is the current user interface (UI) settings for Windows. The UI culture settings determine which text strings appear in user interface elements such as menus and error messages. To determine the current UI culture settings that are active use the Get-UICulture cmdlet. Using Tab Expansion to call the Get-UICulture cmdlet, type the following:

```
Get-Ui + <TAB> + <ENTER>
```

The command and output associated from the command appears here.

```
PS C:\> Get-UICulture


LCID        Name        DisplayName
----        ----        -----------
1033        en-US       English (United States)
```

NOTE : On my laptop, both the current culture and the current UI culture are the same. This is not always the case, and at times, I have seen machines become rather confused when the user interface is set for a localized language, and yet the computer itself was still set for US English (this is especially problematic when using virtual machines created in other countries. In these cases, even a simple task like typing in a password becomes very frustrating. To fix these types of situations you can use the Set-Culture cmdlet.

Finding the current date and time

To find the current date or time on the local computer, use the Get-Date cmdlet. When typing the Get-Date cmdlet name in the Windows PowerShell console tab expansion does not help too much. This is because there are 15 cmdlets (on my laptop) that have a cmdlet name that begins with the letters Get-Da (this includes all of the Direct Access cmdlets as well as the Remote Access cmdlets). Therefore using Tab Expansion (on my laptop anyway) to get the date requires me to type the following:

```
Get-Dat + <TAB> + <Enter>
```

The above command syntax is just the same number of letters to type as doing the following:

```
Get-Date + <ENTER>
```

The following illustrates the command and the output associated with the command.

```
PS C:\> Get-Date
```

Tuesday, November 20, 2012 9:54:21 AM

Generating a random number

Windows 2.0 introduced the Get-Random cmdlet, and when I saw it I was not too impressed. The reason was that I already knew how to generate a random number. Using the .NET Framework System.Random class, all I needed to do was create a new instance of the System.Random object, and call the _next _method. This appears here.

```
PS C:\> (New-Object system.random).next()
225513766
```

Needless to say, I did not create all that many random numbers. I mean, who wants to do all that typing. But once I had the Get-Random cmdlet, I actually began using random numbers for all sorts of things. Some of the things I have used the Get-Random cmdlet to do appear in the following list.

- Pick prize winners for the Scripting Games
- Pick prize winners for Windows PowerShell user group meetings
- To connect to remote servers in a random fashion for load balancing purposes
- To create random folder names
- To create temporary users in active directory with random names
- To wait a random amount of time prior to starting or stopping processes and services (great for performance testing)

The Get-Random cmdlet has turned out to be one of the more useful cmdlets. To generate a random number in the Windows PowerShell console using Tab Expansion type the following on the first line in the console:

```
Get-R +<TAB>+<ENTER>
```

The command, and output associated with the command appears here.

```
PS C:\> Get-Random
248797593
```

Windows PowerShell Basics - Supplying options for cmdlets

The easiest Windows PowerShell cmdlets to use require no options. But unfortunately, that is only a fraction of the total number of cmdlets (and functions) available in Windows PowerShell 4.0 as it exists on either Windows 8.1 or Windows Server 2012 R2. Fortunately, the same Tab Expansion technique used to create the cmdlet names on the Windows PowerShell console, works with **parameters** as well.

Using single parameters

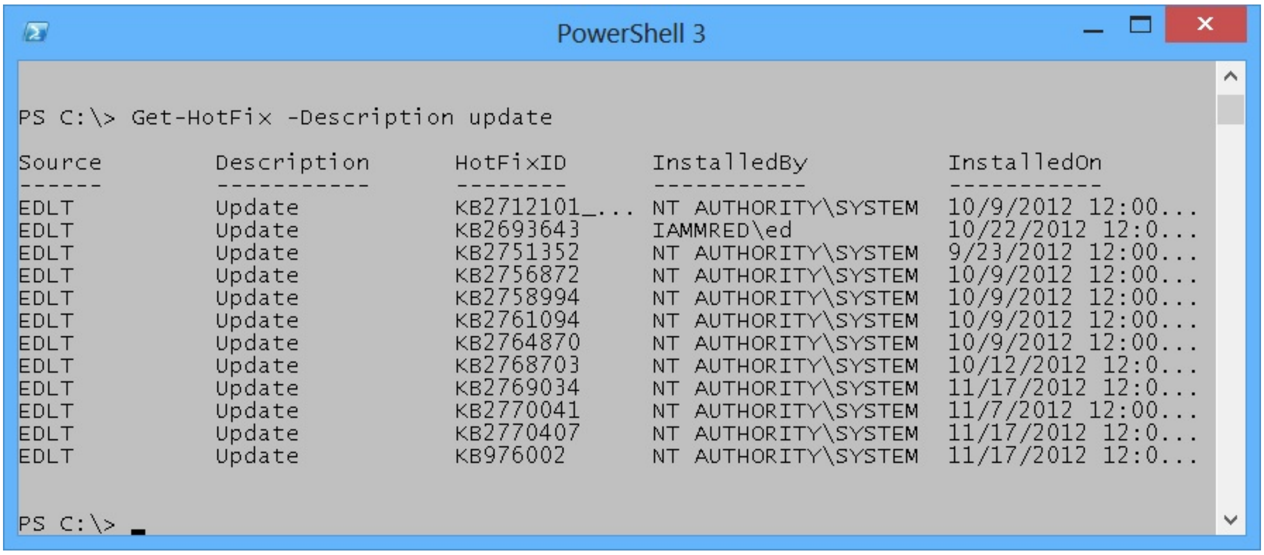
When working with Windows PowerShell cmdlets, often the cmdlet only requires a single parameter to filter out the results. If a parameter is the default parameter, you do not have to specify the parameter name - you can use the parameter positionally. This means that the first value appearing after the cmdlet name, is assumed to be a value for the default (or position 1) parameter. On the other hand, if a parameter is a **named parameter** the parameter name (or parameter alias or partial parameter name) is always required when using the parameter.

Finding specific types of hotfixes

For example to find all of the _update _hotfixes, use the Get-HotFix cmdlet with the -Description parameter and supply a value of _update _to the -Description parameter. This is actually easier than it sounds. Once you type Get-Hot and press the key you have the Get-Hotfix portion of the command. Then a space and -D completes the Get-HotFix -Description portion of the command. Now you need to type Update and press . With a little practice, using Tab Expansion becomes second nature. You only need to type the following:

```
Get-Hot + <TAB> + -D + <TAB> + Update + <ENTER>
```

The completed command and the output associated with the command appear in the figure that follows.



```
PS C:\> Get-HotFix -Description update

Source        Description      HotFixID      InstalledBy          InstalledOn
------        -----------      --------      -----------          -----------
EDLT          Update           KB2712101_... NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Update           KB2693643     IAMMRED\ed           10/22/2012 12:0...
EDLT          Update           KB2751352     NT AUTHORITY\SYSTEM  9/23/2012 12:00...
EDLT          Update           KB2756872     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Update           KB2758994     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Update           KB2761094     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Update           KB2764870     NT AUTHORITY\SYSTEM  10/9/2012 12:00...
EDLT          Update           KB2768703     NT AUTHORITY\SYSTEM  10/12/2012 12:0...
EDLT          Update           KB2769034     NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT          Update           KB2770041     NT AUTHORITY\SYSTEM  11/7/2012 12:00...
EDLT          Update           KB2770407     NT AUTHORITY\SYSTEM  11/17/2012 12:0...
EDLT          Update           KB976002      NT AUTHORITY\SYSTEM  11/17/2012 12:0...

PS C:\> _
```

If you attempt to find only update types of hotfixes by supplying the value _update _in the first position, an error raises. The offending command, and associated error, appears here.

```
PS C:\> Get-HotFix update
Get-HotFix : Cannot find the requested hotfix on the 'localhost' computer. Verify
the input and run the command again.
At line:1 char:1
+ Get-HotFix update
+ ~~~~~~~~~~~~~~~~~
  + CategoryInfo     : ObjectNotFound: (:) [Get-HotFix], ArgumentException
  + FullyQualifiedErrorId : GetHotFixNoEntriesFound,Microsoft.PowerShell.Commands
  .GetHotFixCommand
```

The error, while not really clear, seems to indicate that the Get-HotFix cmdlet attempts to find a hotfix named _update. _This is, in fact, the attempted behavior. The help file information for the Get-HotFix cmdlet reveals that -ID is position 1. This appears here.

```
  -Id <String[]>
    Gets only hotfixes with the specified hotfix IDs. The default is all
    hotfixes on the computer.


    Required?           false
    Position?           1
    Default value         All hotfixes
    Accept pipeline input?    false
    Accept wildcard characters? False
```

Well, what about using the -Description parameter, you may ask? The help file tells that the -Description parameter is a named parameter. This means you can only use the -Description parameter if you specify the parameter name as was accomplished earlier in this section. Here is the applicable portion of the help file for the -Description parameter.

```
  -Description <String[]>
    Gets only hotfixes with the specified descriptions. Wildcards are
    permitted. The default is all hotfixes on the computer.


    Required?          false
    Position?          named
    Default value        All hotfixes
    Accept pipeline input?    false
    Accept wildcard characters? True
```

Finding specific processes

To find process information about a single process, I use the -Name parameter. Because the -Name parameter is the default (position 1) parameter for the Get-Process cmdlet, you do not have to specify the -Name parameter when calling Get-Process if you do not wish to do so. For example, to find information about the PowerShell process by using the Get-Process cmdlet type the following command in the first line of the Windows PowerShell console by using Tab Expansion:

```
Get-Pro + <TAB> + <SPACE> + Po + <TAB> + <ENTER>
```

The completed command and associated output appears here.

```
PS C:\> Get-Process powershell


Handles NPM(K)  PM(K)   WS(K) VM(M)  CPU(s)   Id ProcessName
------- ------  -----   ----- -----  ------   -- -----------
  607   39  144552   164652  718   5.58  4860 powershell
```

You can tell that the Get-Process cmdlet accepts the -Name parameter in a positional manner because the Help file states it is in position 1. This appears here.

```
  -Name <String[]>
    Specifies one or more processes by process name. You can type multiple
    process names (separated by commas) and use wildcard characters. The
    parameter name ("Name") is optional.


    Required?          false
    Position?          1
    Default value
    Accept pipeline input?    true (ByPropertyName)
    Accept wildcard characters? True
```

NOTE : Be careful using positional parameters. This is because they can be confusing. For example, the first parameter for the Get-Process cmdlet is the -Name parameter, but the first position parameter for the Stop-Parameter is the -ID parameter. As a best practice always refer to the Help files to see what the parameters actually are called, and the position in which they are expected. This is even more important when using cmdlet with multiple parameters - such as the Get-Random cmdlet discussed next.

Generating random numbers in a range

When used without any parameters, the Get-Random cmdlet returns a number that is in the range of 0 to 2,147,483,647. We have never had a Windows PowerShell user group meeting in which there were either 0 people in attendance, nor have we had a Windows PowerShell user group meeting with 2,147,483,647 people in attendance. Therefore when handing out prizes at the end of the day, it is important to set a different minimum and maximum number.

NOTE : When using the -Maximum parameter for the Get-Random cmdlet keep in mind that the maximum number never appears. Therefore, if you have 15 people attending your Windows PowerShell user group meeting, you would want to set the -Maximum parameter to 16 (unless you do not like the 15 person and do not want them to win any prizes).

The default parameter for the Get-Random cmdlet is the -Maximum parameter. This means that you can use the Get-Random cmdlet to generate a random number in the range of 0 to 20 by using Tab Expansion on the first line of the Windows PowerShell console. Type the following (remember Get-Random never reaches the maximum number, therefore always use a number 1 greater than the desired upper number):

```
Get-R + <TAB> + <SPACE> + 21
```

If you want to generate a random number between 1 and 20, you might think you could use Get-Random 1 21, but that generates an error. The command and the error appear here.

```
PS C:\> Get-Random 1 21
Get-Random : A positional parameter cannot be found that accepts argument '21'.
At line:1 char:1
+ Get-Random 1 21
+ ~~~~~~~~~~~~~~~
  + CategoryInfo     : InvalidArgument: (:) [Get-Random], ParameterBindingEx
  ception
  + FullyQualifiedErrorId : PositionalParameterNotFound,Microsoft.PowerShell.Comm
  ands.GetRandomCommand
```

The error states that *a positional parameter cannot be found that accepts argument '21'*. This is because the Get-Random only has one positional parameter - the -Maximum parameter. The -Minimum parameter is a named parameter (this appears in the Help file for the Get-Random cmdlet. Use of the Help files appears in Chapter two).

To generate a random number in the range of 1 to 20, use named parameters. To assist in creating the command use Tab Expansion for the cmdlet name as well as for the parameter names. Type the following to create the command using Tab Expansion.

```
Get-R + <TAB> + -M + <TAB> + <SPACE> + 21 + -M + <TAB> + <SPACE> + 1 + <ENTER>
```

The command and the output associated with the command appears here.

```
PS C:\> Get-Random -Maximum 21 -Minimum 1
19
```

An introduction to parameter sets

One of the things that quickly becomes confusing with Windows PowerShell cmdlets is that there are often different ways of using the same cmdlet. For example, you can specify the -Minimum and the -Maximum parameters, but you cannot also specify the -Count parameter. This is a bit unfortunate, because it would seem that using the -Minimum and the -Maximum parameters to specify the minimum and the maximum numbers for the random numbers makes sense. When the Windows PowerShell user group has five prizes to give away it is inefficient to have to either write a script to generate the five random numbers. It is also inefficient to have to run the same command five times.

This is where command sets come into play. The -Minimum and the -Maximum parameters specify the range within which to pick a single random number. To generate more than one random number use the -Count parameter. Here are the two parameter sets.

```
Get-Random [[-Maximum] <Object>] [-Minimum <Object>] [-SetSeed <Int32>]
[<CommonParameters>]


Get-Random [-InputObject] <Object[]> [-Count <Int32>] [-SetSeed <Int32>]
[<CommonParameters>]
```

The first parameter set accepts -Maximum, -Minimum and -SetSeed. The second parameter set accepts -InputObject, -Count and -SetSeed. Therefore you cannot use -Count with -Minimum or -Maximum - they are in two different groups of parameters (called parameter sets).

NOTE : It is quite common for Windows PowerShell cmdlets to have multiple parameter sets. Tab Expansion only offers parameters from one parameter set - therefore when you choose a parameter (such as -Count from Get-Random) the non-compatable parameters do not appear in tab Expansion. This feature keeps you from creating invalid commands. For an overview of a cmdlets parameter sets, use the Get-Help cmdlet.

Generating a certain number of random numbers

The Get-Random cmdlet, when used with the -Count parameter accepts an -InputObject parameter. The -InputObject parameter is quite powerful. The help file, appearing here, states that it accepts a collection of objects.

```
-InputObject <Object[]>
  Specifies a collection of objects. Get-Random gets randomly selected
  objects in random order from the collection. Enter the objects, a variabl
  that contains the objects, or a command or expression that gets the
  objects. You can also pipe a collection of objects to Get-Random.


  Required?          true
  Position?          1
  Default value
  Accept pipeline input?    true (ByValue)
  Accept wildcard characters? False
```

An array (or a range) of numbers just happens to also be a collection of objects. The easiest way to generate a range (or an array) of numbers is to use the range operator. The **range operator** is two dots (periods) between two numbers. The **range operator** does not require spaces between the numbers, and dots. This appears here.

```
PS C:\> 1..5
1
2
3
4
5
```

Now to pick five random numbers from the range of 1 to 10, only requires the command appearing here. (The parentheses are required around the range of 1 to 10 numbers to ensure the range of numbers creates prior to attempting to select five from the collection.

```
Get-Random -InputObject (1..10) -Count 5
```

The command and output associated with the command appear here.

```
PS C:\> Get-Random -InputObject (1..10) -Count 5
7
5
10
1
8
```

Windows PowerShell Basics - Using command line utilities

As easy as Windows PowerShell is to use, there are times when it is easier to find information by using a command line utility. For example, to find IP configuration information you only need to use the _Ipconfig.exe _utility. You can type this directly into the Windows PowerShell console and read the output in the Windows PowerShell console. This command and associated output appears here in truncated form.

```
PS C:\> ipconfig

Windows IP Configuration

Wireless LAN adapter Local Area Connection\* 14:

 Media State . . . . . . . . . . . : Media disconnected
 Connection-specific DNS Suffix . :

Ethernet adapter vEthernet (WirelessSwitch):

 Connection-specific DNS Suffix . : quadriga.com
 Link-local IPv6 Address . . . . . : fe80::915e:d324:aa0f:a54b%31
 IPv4 Address. . . . . . . . . . . : 192.168.13.220
 Subnet Mask . . . . . . . . . . . : 255.255.248.0
 Default Gateway . . . . . . . . . : 192.168.15.254

Wireless LAN adapter Local Area Connection\* 12:

 Media State . . . . . . . . . . . : Media disconnected
 Connection-specific DNS Suffix . :

Ethernet adapter vEthernet (InternalSwitch):

 Connection-specific DNS Suffix . :
 Link-local IPv6 Address . . . . . : fe80::bd2d:5283:5572:5e77%19
 IPv4 Address. . . . . . . . . . . : 192.168.3.228
 Subnet Mask . . . . . . . . . . . : 255.255.255.0
 Default Gateway . . . . . . . . . : 192.168.3.100

<OUTPUT TRUNCATED>
```

To obtain the same information using Windows PowerShell would require a more complex command. The command to obtain IP information is Get-NetIPAddress, But there are several advantages. For one thing, the output from the _IpConfig.exe _command is text,

whereas the output from Windows PowerShell is an object. This means you can group, sort, filter, and format the output in an easy fashion.

The cool thing is that with Windows PowerShell console, you have not only the simplicity of the command prompt, but you also have the powerful Windows PowerShell language built in. Therefore, if you need to refresh Group Policy three times and wait for five minutes between refreshes, you can use the command appearing here (looping is covered in chapter eleven).

```
1..3 | % {gpupdate ; sleep 300}
```

Windows PowerShell Basics - Working with help options

The first thing you need to do is to update the help files on your system. This is because Windows PowerShell 3.0 introduces a new model in which the help files update on a regular basis.

To update help on your system, you must ensure two things. The first is that you open the Windows PowerShell console with ADMIN rights. This is because the Windows PowerShell help files reside in the protected Windows\System32\WindowsPowerShell directory. Once you have launched the Windows PowerShell console with admin rights you need to ensure your computer has Internet access so it can download and install the updated files. If your computer does not have Internet connectivity, it will take several minutes before the command times out (Windows PowerShell tries really hard to obtain the updated files). If you run the Update-Help cmdlet with no parameters Windows PowerShell attempts to download updated help for all modules stored in the default Windows PowerShell modules locations that support updatable help. To run Update-Help more than once a day use the -Force parameter as appears here.

```
Update-Help -Force
```

Even without downloading updated Windows PowerShell help, the help subsystem displays the syntax of the cmdlet and other rudimentary information about the cmdlet. In this way.

To display help information from the internet, use the -Online switch. When used in this way, Windows PowerShell causes the default browser to open to the appropriate page from the Microsoft TechNet web site.

In the enterprise, network administrators may want to use the Save-Help cmdlet to download help from the Internet. Once downloaded, the Update-Help cmdlet can point to the network share for the files. This is an easy task to automate, and can run as a scheduled task.

Windows PowerShell Basics - Working with modules

What makes the big difference in capabilities between Windows PowerShell 4.0 installed on Windows 7 or Windows 8.1 is not the difference in the capability of Windows PowerShell 4.0. The package provides the same abilities. The difference is the modules introduced with Windows 8 and expanded in Windows 8.1. To find out the commands that a module provides, I use the Get-Command cmdlet and specify the name of a particular module. In this example, I look at the commands provided by the NetAdapter.

```
Get-Command-ModuleNetAdapter
```

If I use the Get-Command cmdlet and an error arises, it may be because the module has not yet loaded. To load the module use the Import-Module cmdlet. This command appears here.

```
Import-ModuleNetAdapter
```

If I am curious as to the number of commands exposed by the module, I can pipeline the results to the Measure-Object cmdlet. This command appears here.

```
Get-Command-Modulenetadapter|Measure-Object
```

Working with Network adapters

Windows offers many different ways to work with Network Adapters. The correct choice depends upon several things. First probably, I need to know what version of the operating system I am running. In most cases the version of the operating system will either limit or expand my options for working with Network Adapters. Next I need to know if I am working locally or remotely, because where I run my commands from often determine my choice of tool. Lastly I always choose the tool that is easiest for me to do the job I have to perform. This is not always the easiest tool for anyone to use, but I choose the tool that I know. For me, for example, typing even a dozen commands into the Windows PowerShell ISE is much easier than attempting to use NetSh in some context with which I am unfamiliar. In addition, by typing my commands into the Windows PowerShell ISE, I can easily save my commands off as a Windows PowerShell script, that I can reuse. Of course I can reuse NetSh commands - I do all the time, but it is an extra step. So, to summarize, what is my decision matrix (assuming identical capabilities)?

1. Version of Operating System
2. Remoting capability
3. Ease of use

All things being equal, what tools are available to me to use to accomplish my work with network adapters?

1. Windows PowerShell
2. NetSH
3. Windows Management Instrumentation (WMI)
4. VBScript
5. Console Utilities

In this booklet, I will talk about each of these approaches as I look at the different tasks. So what tasks am I talking about? Well, I am specifically talking about the network adapter. So here are the things I am going to cover:

1. Identifying network adapters
2. Enabling and disabling network adapters
3. Renaming network adapters
4. Finding connected network adapters
5. Identifying network adapter power setting
6. Configuring network adapter power settings
7. Gathering network adapter statistics

Along the way, I will be showing some pretty cool Windows PowerShell tricks.

PowerTip : Find protocol binding on net adapters using PowerShell

Question: How can you use Windows PowerShell to show which enabled protocols are bound to your network adapters using Windows 8.1 and PowerShell 4.0?

Answer: Use the Get-Netadapter cmdlet to retrieve all of the network adapters on your system. Then pipeline it to the Get-NetAdapterBinding cmdlet and filter on enabled is equal to true. This command appears here:

```
Get-NetAdapter|Get-NetAdapterBinding|-enabled-eq$true
```

Identifying network adapters

One of the great things about Windows Management Instrumentation (WMI) is the way that it can provide detailed information. The bad thing is that it requires a specialist level of knowledge and understanding to effectively use and to understand the information (either that or a good search engine, such as BING and an awesome repository of information such as the Script Center).

Using raw WMI to identify network adapters

One of the cool things about Windows PowerShell, since version 1.0, is that it provides easier access to WMI information. The bad thing, of course, is that it is still wrestling with WMI, which some IT Pro's seem to hate (or at least dislike). The great thing about using raw WMI is compatibility with older versions of the operating system. For example, using raw WMI and Windows PowerShell would make it possible to talk to Windows XP, Windows 2003 Server, Windows 2008 Server, Vista, Windows Server 2008 R2 and Windows 7, in addition to the modern operating systems of Windows 8, 8.1 and Windows Server 2012 and Windows Server 2012 R2.

So how do I do it? I used to be able to find our real network card by finding the one that was bound to TCP/IP. I would query the Win32_NetworkAdapterConfiguration WMI class, and filter on the IPEnabled property. Using this approach, I would have done something like this:

```
Get-WmiObject-ClassWin32\_NetworkAdapterConfiguration -filter"IPEnabled = $true"
```

The problem with this methodology nowadays is that some of the pseudo adapters are also IPEnabled. The above command would eliminate many, but not necessarily all of the adapters.

A better approach is to look at the Win32_NetworkAdapter class and query the NetConnectionStatus property. Using this technique, I return only network adapter devices that are actually connected to a network. While it is possible that a pseudo adapter could sneak under the wire, the likelihood is more remote. In this command, I will use the Get-WmiObject PowerShell cmdlet to return all instances of Win32_NetworkAdapter class on the computer. I then create a table to display the data returned by the NetConnectionStatus property.

```
Get-WmiObject-ClassWin32\_NetworkAdapter|
Format-Table-PropertyName,NetConnectionStatus-AutoSize
```

The fruit of our labor is somewhat impressive. I have a nice table that details all of the fake and real network adapters on our laptop, as well as the connection status of each. Here is the list from my laptop.

```
Name                       NetConnectionStatus

----                       -------------------

WAN Miniport (L2TP)
WAN Miniport (PPTP)
WAN Miniport (PPPOE)
WAN Miniport (IPv6)
Intel(R) PRO/1000 PL Network Connection     2
Intel(R) PRO/Wireless 3945ABG Network Connection 0
WAN Miniport (IP)
Microsoft 6to4 Adapter
Bluetooth Personal Area Network
RAS Async Adapter
isatap.{51AAF9FF-857A-4460-9F17-92F7626DC420}
Virtual Machine Network Services Driver
Microsoft ISATAP Adapter
Bluetooth Device (Personal Area Network)     7
6TO4 Adapter
Microsoft 6to4 Adapter
Microsoft Windows Mobile Remote Adapter
isatap.launchmodem.com
isatap.{647A0048-DF48-4E4D-B07B-2AE0995B269F}
Microsoft Windows Mobile Remote Adapter
WAN Miniport (SSTP)
WAN Miniport (Network Monitor)
6TO4 Adapter
6TO4 Adapter
Microsoft 6to4 Adapter
Microsoft Windows Mobile Remote Adapter
isatap.{C210F3A1-6EAC-4308-9311-69EADBA00A04}
isatap.launchmodem.com
Virtual Machine Network Services Driver
Virtual Machine Network Services Driver
Teredo Tunneling Pseudo-Interface
isatap.{647A0048-DF48-4E4D-B07B-2AE0995B269F}
```

There are two things you will no doubt notice. The first is that most of the network adapters report no status what-so-ever. The second thing you will notice is that the ones that do report a status do so in some kind of code. The previous table is therefore pretty much useless! But it does look nice.

A little work in the Windows SDK looking up the Win32_NetworkAdapter WMI class and I run across the following information:

```
Value Meaning

0     Disconnected

1     Connecting

2     Connected

3     Disconnecting

4     Hardware not present

5     Hardware disabled

6     Hardware malfunction

7     Media disconnected

8     Authenticating

9     Authentication succeeded

10    Authentication failed

11    Invalid address

12    Credentials required
```

The value of 2 means the network adapter is connected. Here is the code I wrote to exploit the results of our research.

```
Get-WmiObject -classwin32\_networkadapter -filter "NetConnectionStatus = 2"|
Format-List -Property [a-z]\*
```

Such ecstasy is short lived; however, when I realize that while I have indeed returned information about a network adapter that is connected, I do not have any of the configuration information from the card.

What I need is to be able to use the NetConnectionStatus property from Win32_Networkadapter and to be able to obtain the Tcp/Ip configuration information from the Win32_NetworkAdapterConfiguration WMI class. This sounds like a job for an association class. In VBScript querying an Association class involved performing confusing AssociatorsOf queries (Refer to the MSPress book, " Window Scripting with WMI: Self Paced Learning Guide" for more information about this technique.)

Using the association class with Windows PowerShell, I come up with the FilterAssociatedNetworkAdapters.ps1 script shown here.

FilterAssociatedNetworkAdapters.ps1

```
Param($computer="localhost")

functionfunline ($strIN)
{
$num=$strIN.length
for($i=1 ; $i-le$num ; $i++)
 { $funline=$funline+"=" }
  Write-Host-ForegroundColoryellow$strIN
  Write-Host-ForegroundColordarkYellow$funline
} #end funline

Write-Host-ForegroundColorcyan"Network adapter settings on $computer"

Get-WmiObject -Class win32\_NetworkAdapterSetting `
-computername $computer|
Foreach-object `
{
 If( ([wmi]$\_.element).netconnectionstatus -eq 2)
  {
  funline("Adapter: $($\_.setting)")
  [wmi]$\_.setting
  [wmi]$\_.element
  } #end if
} #end foreach
```

I begin the script by using a command line parameter to allow us to run the script remotely if needed. I use the Param statement to do this. I also create a function named funline that is used to underline the results of the query. It makes the output nicer if there is a large amount of data returned.

```
Param($computer="localhost")

functionfunline ($strIN)
{
$num=$strIN.length
for($i=1 ; $i-le$num ; $i++)
 { $funline=$funline+"=" }
  Write-Host-ForegroundColoryellow$strIN
  Write-Host-ForegroundColordarkYellow$funline
} #end funline
```

I print out the name of the computer by using the Write-Host cmdlet as seen here. I use the color cyan so the text will show up real nice on the screen (unless of course your background is also cyan, then the output will be written in invisible ink. That might be cool as well.)

```
Write-Host -Foreground Colorcyan "Network adapter settings on $computer"
```

Then I get down to actual WMI query. To do this, I use the Get-WmiObject cmdlet. I use the -computername parameter to allow the script to run against other computers, and I pipeline the results to the ForEach-Object cmdlet.

```
Get-WmiObject -Class win32\_NetworkAdapterSetting `
-computername $computer|
Foreach-object `
```

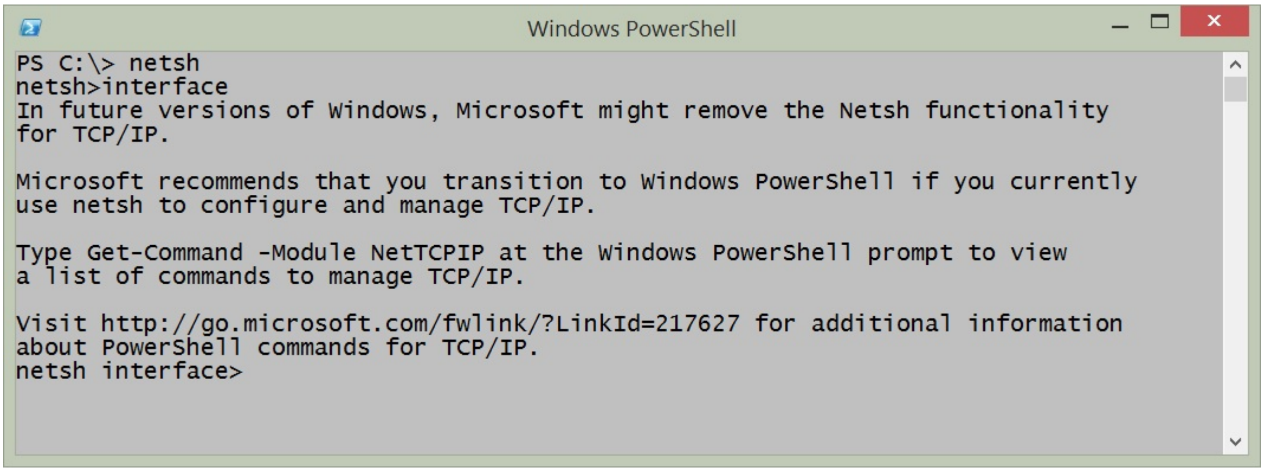
The hard part of the query is seen here. I need a way to look at the netConnectionStatus property of the Win32_NetworkAdapter class. This class is referred to by the reference returned from the association query. It is called element. To gain access to this class, I use the reference that was returned and feed it to the [WMI] type accelerator (it likes to receive a path, and this is what the reference is). Since the reference refers to a specific instance of a WMI class, and since the [WMI] type accelerator can query a specific instance of a class, I are now able to obtain the value of the netConenctionStatus property. So I say in our script, if it is equal to 2, then I will print out the name of the network adapter, and the configuration that is held in the setting property and the adapter information that held in the element property. This section of the code is seen here.

```
{
 If( ([wmi]$\_.element).netconnectionstatus -eq2)
  {
  funline("Adapter: $($\_.setting)")
  [wmi]$\_.setting
  [wmi]$\_.element
  } #end if
```

The result of running the script is that it displays information from both the Win32_NetworkAdapter WMI class and the Win32_NetworkAdapterConfiguration class. It also shows us I only have one connected network adapter.

Using NetSh

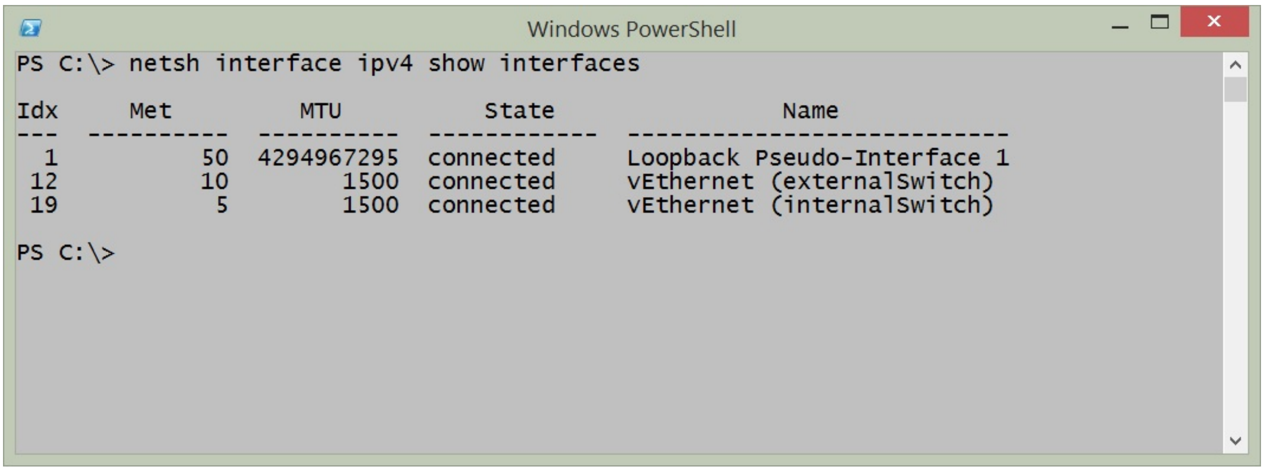
Microsoft created NetSh back in 2000, and it has been a staple of networking ever since then. When I open it up, now days, it displays a message saying that it might be removed in future versions of Windows, and therefore I should begin using Windows PowerShell. Here is the message:



Now, because NetSh is an old style menu type application, it is possible to enter NetSh, and walk my way down through the menus until you arrive at the proper location. Along the way, if I get lost, I can use the ? to obtain help. The problem, is that the help is quite often not very helpful, and therefore it takes me at times nearly a dozen times before the command is correct. The great thing is that, for the most part, Once I figure out a command, I can actually keep track of my location in the program, and back all the way out and enter the command as a one linner. Here is the NetSh command to display network interface information that is bound to Ipv4:

```
netsh interface ipv4 show interfaces
```

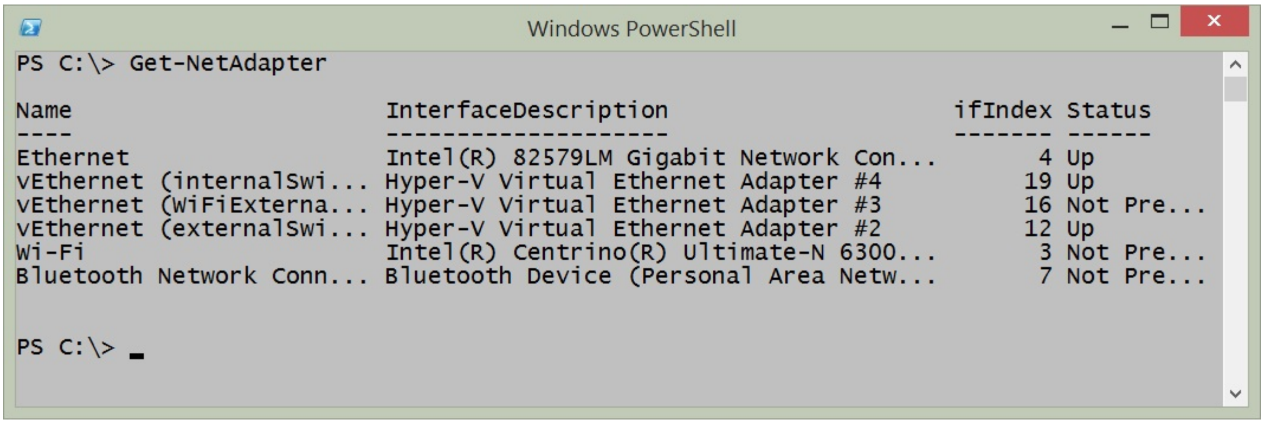
The output appears here:



Using PowerShell on Windows 8 or above

If I have the advantage of Windows 8 or 8.1 or Windows Server 2012 or Windows Server 2012 R2, then I have the built in NetAdapter module. Due to the way that modules autoload on Windows Powell I do not need to remember that I am using functions that exist in the NetAdapter module. I can use either Windows PowerShell 3 or Windows PowerShell 4 and the behavior will be the same (Windows 8.1 and Windows Server 2012 R2 come with Windows PowerShell 4 and Windows 8 and Windows Server 2012 come with Windows PowerShell 3).

The Get-NetAdapter cmdlet returns the name, interface description, index number, and status of all network adapters present on the system. This is the default display of information and appears in the figure that follows.



To focus in on a particular network adapter, I use the _name _parameter and supply the name of the network adapter. The good thing, is that in Windows 8 (and on Windows Server 2012) the network connections receive new names. No more of the "local area connection" and "local area connection(2) to attempt to demystify. The wired network adapter is simply _Ethernet _and the wireless network adapter is _Wi-Fi. _The following command retrieves only then _Ethernet _network adapter.

```
Get-NetAdapter -Name Ethernet
```

To dive into the details of the _Ethernet _network adapter, I pipeline the returned object to the Format-List cmdlet and I choose all of the properties. The command appearing here uses the _fl _alias for the Format-List cmdlet.

```
Get-NetAdapter -Name ethernet|Format-List *
```

The command and output associated with the command appear in the figure that follows.

```
Windows PowerShell
PS C:\> Get-NetAdapter -name ethernet | fl *


ifAlias                                          : Ethernet
InterfaceAlias                                   : Ethernet
ifIndex                                          : 4
ifDesc                                           : Intel(R) 82579LM Gigabit Network
                                                   Connection
ifName                                           : Ethernet_0
DriverVersion                                    : 12.2.46.0
LinkLayerAddress                                 : 3C-97-0E-75-BD-88
MacAddress                                       : 3C-97-0E-75-BD-88
Status                                           : Up
LinkSpeed                                        : 1 Gbps
MediaType                                        : 802.3
PhysicalMediaType                                : 802.3
AdminStatus                                      : Up
MediaConnectionState                             : Connected
DriverInformation                                : Driver Date 2012-11-26 Version
                                                   12.2.46.0 NDIS 6.30
DriverFileName                                   : e1c63x64.sys
NdisVersion                                      : 6.30
ifOperStatus                                     : Up
Caption                                          :
Description                                      :
ElementName                                      :
InstanceID                                       : {CE652A65-65BB-4A83-AAF6-53077EB1
                                                   B6CD}
CommunicationStatus                              :
DetailedStatus                                   :
HealthState                                      :
InstallDate                                      :
Name                                             : Ethernet
OperatingStatus                                  :
OperationalStatus                                :
PrimaryStatus                                    :
StatusDescriptions                               :
AvailableRequestedStates                         :
EnabledDefault                                   : 2
EnabledState                                     : 5
OtherEnabledState                                :
RequestedState                                   : 12
TimeOfLastStateChange                            :
```

There are a number of excellent properties that might bear further investigation, for example there are the _adminstatus _and the _mediaconnectionstatus _properties. The following command returns those two properties.

```
Get-NetAdapter -Name ethernet |select adminstatus, MediaConnectionState
```

Of course, there are other properties that might be interesting as well. These properties appear here, along with the associated output (the following is a single logical command broken on two lines).

```
Get-NetAdapter -Name ethernet |
select ifname, adminstatus, MediaConnectionState, LinkSpeed, PhysicalMediaType
```

The output from the above command appears here:

```
ifName        : Ethernet\_7
AdminStatus     : Down
MediaConnectionState : Unknown
LinkSpeed      : 0 bps
PhysicalMediaType  : 802.3
```

I decide to look only for network adapters that are in the admin status of _up. _I use the command appearing here.

```
PS C:\> Get-NetAdapter|whereadminstatus-eq"up"

Name           InterfaceDescription          ifIndex Status
----           --------------------          ------- ------
vEthernet (InternalSwi... Hyper-V Virtual Ethernet Adapter #3     22 Up
vEthernet (ExternalSwi... Hyper-V Virtual Ethernet Adapter #2     19 Up
Bluetooth Network Conn... Bluetooth Device (Personal Area Netw...   15 Disconn...
Wi-Fi            Intel(R) Centrino(R) Ultimate-N 6300...   12 Up
```

To find the disabled network adapters, I change the _adminstatus _from _up _to _down. _This command appears here.

```
Get-NetAdapter|whereadminstatus-eq"down"
```

I go back to my previous command, and modify it to return WI-FI information. This command, and associated output appears here (this is a single logical command).

```
PS C:\> Get-NetAdapter-Name wi-fi|
select ifname,adminstatus,MediaConnectionState,LinkSpeed,PhysicalMediaType

ifName        : WiFi\_0
AdminStatus     : Up
MediaConnectionState : Connected
LinkSpeed      : 54 Mbps
PhysicalMediaType  : Native 802.11
```

If I want to find any network adapters sniffing the network, I look for _promiscousmode. _This command appears here.

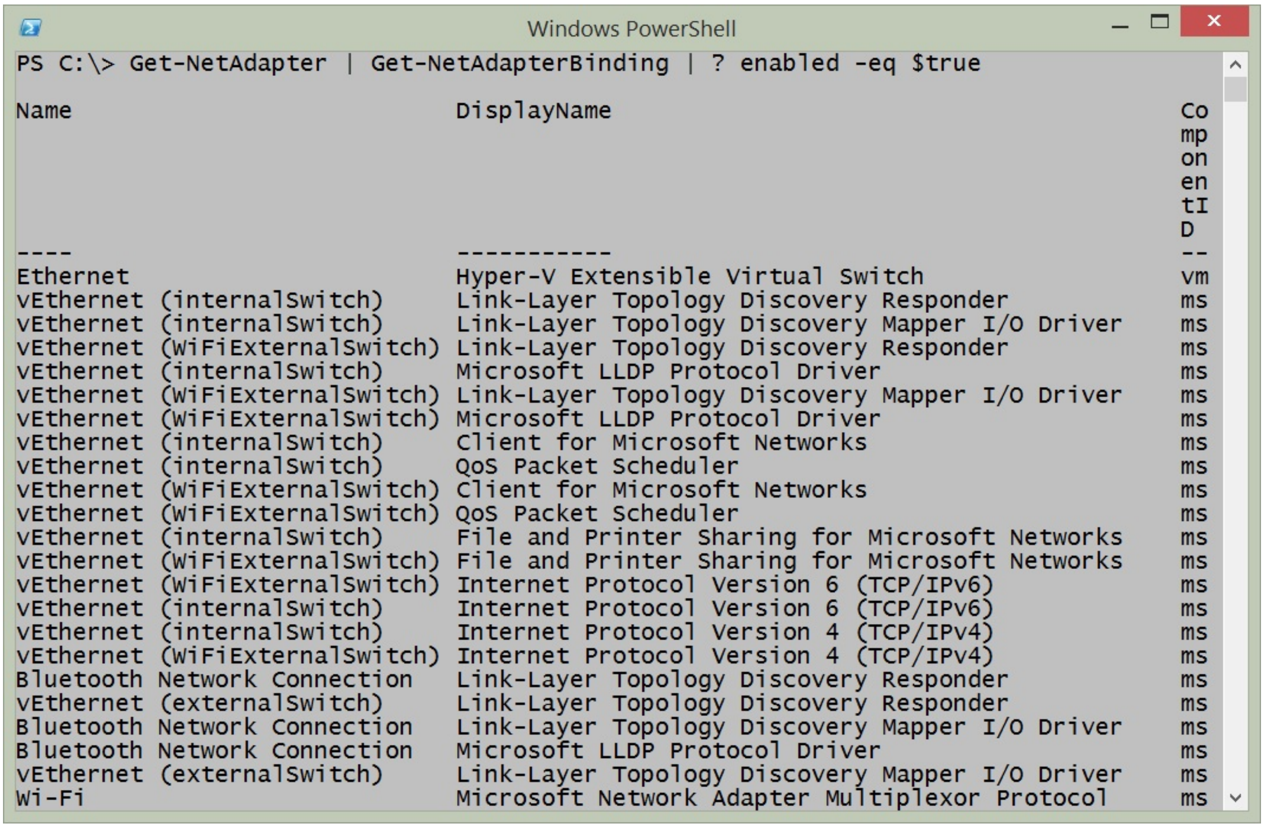
```
Get-NetAdapter| ? PromiscuousMode -eq $true
```

When I combine the Get-NetAdapter function with the Get-NetAdapterBinding function I can easily find out which protocols are bound to which network adapter. I just send the results to the Where-Object and check to see if the enabled property is equal to true or not. Here is the

command.

```
Get-NetAdapter |Get-NetAdapterBinding | ? enabled -eq $true
```

Here is an example of both the command and the output from the command.



```
Windows PowerShell
PS C:\> Get-NetAdapter | Get-NetAdapterBinding | ? enabled -eq $true

Name                           DisplayName                                        Co
                                                                                  mp
                                                                                  on
                                                                                  en
                                                                                  tI
                                                                                  D
----                           -----------                                        --
Ethernet                       Hyper-V Extensible Virtual Switch                  vm
vEthernet (internalSwitch)     Link-Layer Topology Discovery Responder            ms
vEthernet (internalSwitch)     Link-Layer Topology Discovery Mapper I/O Driver    ms
vEthernet (WiFiExternalSwitch) Link-Layer Topology Discovery Responder            ms
vEthernet (internalSwitch)     Microsoft LLDP Protocol Driver                     ms
vEthernet (WiFiExternalSwitch) Link-Layer Topology Discovery Mapper I/O Driver    ms
vEthernet (WiFiExternalSwitch) Microsoft LLDP Protocol Driver                     ms
vEthernet (internalSwitch)     Client for Microsoft Networks                      ms
vEthernet (internalSwitch)     QoS Packet Scheduler                               ms
vEthernet (WiFiExternalSwitch) Client for Microsoft Networks                      ms
vEthernet (WiFiExternalSwitch) QoS Packet Scheduler                               ms
vEthernet (internalSwitch)     File and Printer Sharing for Microsoft Networks    ms
vEthernet (WiFiExternalSwitch) File and Printer Sharing for Microsoft Networks    ms
vEthernet (WiFiExternalSwitch) Internet Protocol Version 6 (TCP/IPv6)             ms
vEthernet (internalSwitch)     Internet Protocol Version 6 (TCP/IPv6)             ms
vEthernet (internalSwitch)     Internet Protocol Version 4 (TCP/IPv4)             ms
vEthernet (WiFiExternalSwitch) Internet Protocol Version 4 (TCP/IPv4)             ms
Bluetooth Network Connection   Link-Layer Topology Discovery Responder            ms
vEthernet (externalSwitch)     Link-Layer Topology Discovery Responder            ms
Bluetooth Network Connection   Link-Layer Topology Discovery Mapper I/O Driver    ms
Bluetooth Network Connection   Microsoft LLDP Protocol Driver                     ms
vEthernet (externalSwitch)     Link-Layer Topology Discovery Mapper I/O Driver    ms
Wi-Fi                          Microsoft Network Adapter Multiplexor Protocol     ms
```

If I want to find which network adapters have the Client for Microsoft Networks bound, I need to first see which protocols are enabled (using the syntax from the previous command) and I need to see which one of the enabled protocols have the display name of Client for Microsoft Networks. This requires a compound where-object statement and therefore I cannot use the simplified syntax. Also, because only one of the protocols begins with Client - I can use that to shorten my query just a bit. Here is the command I use (this is a one line command that I broke at the pipe character to make a better display).

```
Get-NetAdapter |
Get-NetAdapterBinding |
where {$\_.enabled -AND$\_.displayname -match 'client'}
```

The command and associated output appear in the figure here.

```
PS C:\> Get-NetAdapter | Get-NetAdapterBinding | where {$_.enabled -AND $_.displaynam
e -match 'client'}

Name                          DisplayName                                          Co
                                                                                   mp
                                                                                   on
                                                                                   en
                                                                                   tI
                                                                                   D
----                          -----------                                          --
vEthernet (internalSwitch)    Client for Microsoft Networks                        ms
Bluetooth Network Connection  Client for Microsoft Networks                        ms
vEthernet (externalSwitch)    Client for Microsoft Networks                        ms
vEthernet (WiFiExternalSwitch) Client for Microsoft Networks                       ms


PS C:\>
```

Enabling and disabling network adapters

One of the most fundamental things that I do with a network adapter is either enable or disable it. In fact, I perform these tasks several times a week. This is because my primary work device is a laptop and it has built-in wireless network adapters. Not surprising, all modern laptops have both wired and wireless connections available. But when I am at home, in my office I want to have my laptop use the gigabit Ethernet switch that I have, and not go through the significantly slower wireless adapter. If I am on the road, I want to know if my wireless network adapter is enabled or not, and I want to control whether it connects to say a network named Starbucks for example. If I do not control such a thing, my laptop will automatically connect to every wireless it has ever seen before. This is why, for example , I wrote this blog article about cleaning out wireless network history. Chris Wu, a Microsoft PFE also wrote an article that takes a different approach. It is a good read as well.

PowerTip : Enable all network adapters

Question: You are troubleshooting your Windows 8.1 laptop and want to quickly enable all network adapters. How can you do this?

Answer: Use the Get-NetAdapter and the Enable-NetAdapter commands. The command line appears here:

```
Get-NetAdapter |Where status -ne up | Enable-NetAdapter
```

Using Devcon

In the old days, back before Windows Vista and Windows Server 2008 when I needed to enable or disable a network adapter, I would actually use Devcon. Devcon is a command line utility that provides the ability to enable and to disable various hardware devices. It is billed as a command-line Device Manager. Here is a VBScript I wrote to enable and to disable the network interface adapter using Devcon. Keep in mind that Devcon is not installed by default, and therefore must be installed prior to use.

```
'==========================================================================
'
' VBScript: AUTHOR: Ed Wilson , MS, 5/5/2004
'
' NAME: <turnONoffNet.vbs>
'
' COMMENT: Key concepts are listed below:
```

```
'1.uses the c:\devcon utility to turn on or off net
'2.uses a vbyesNO msgBOX to solicit input
'3. KB 311272 talks about devcon and shows where to get
'==========================================================================

Option Explicit

Dim objShell
Dim objExec
Dim onWireLess
Dim onLoopBack
Dim turnON
Dim turnOFF
Dim yesNO
Dim message, msgTitle
Dim strText

message = "Turn On Wireless? Loop is disabled" & vbcrlf & "if not, then wireless is disab

msgTitle = "change Network settings"
onWireLess = " PCMCIA\Dell-0156-0002"
onLoopBack = " \*loop"
turnON = "enable"
turnOFF = "disable"
Const yes = 6
Set objShell = CreateObject("wscript.shell")
yesNO = MsgBox(message,vbyesNO,msgTitle)

If yesNO = yes Then
WScript.Echo "yes chosen"
Set objExec = objShell.exec("cmd /c c:\devcon " & turnON & onWireLess)
subOUT
Set objExec = objShell.exec("cmd /c c:\devcon " & turnOFF & onLoopBack)
subOUT
Else
WScript.Echo "no chosen"
Set objExec = objShell.exec("cmd /c c:\devcon " & turnOFF & onWireLess)
subOUT
Set objExec = objShell.exec("cmd /c c:\devcon " & turnON & onLoopBack)
subOUT
End If

Sub subOUT
Do until objExec.StdOut.AtEndOfStream
  strText = objExec.StdOut.ReadLine()
  Wscript.Echo strText
Loop
End sub
```

Using WMI

Beginning with Windows Vista (and Windows Server 2008) the Win32_NetworkAdapter class gains two methods: disable and enable. These are documented on MSDN here. These methods are instance methods which means that to use them, I need to first obtain an instance of the WMI class. What does this mean? Well I am using Win32_NetworkAdapter and therefore I am working with network adapters. So, I need to get a specific network adapter, and then I can disable it or enable it. Here is how it might work:

```
$wmi=Get-WmiObject-ClassWin32\_NetworkAdapter-filter"Name LIKE '%Wireless%'"

$wmi.disable()
```

OR

```
$wmi=Get-WmiObject-ClassWin32\_NetworkAdapter-filter"Name LIKE '%Wireless%'"

$wmi.enable()
```

The thing to keep in mind is that when calling a method in Windows PowerShell, the parenthesis are required.

If I need to specify alternate credentials, I can specify a remote computer name and an account that has local admin rights on the remote box. The code would appear like the following:

```
$wmi=Get-WmiObject-ClassWin32\_NetworkAdapter-filter"Name LIKE '%Wireless%'"-credential (
$wmi.disable()
```

Keep in mind that WMI does not permit alternate credentials for a local connection. Attempts to use alternate credentials for a local connection results in the error appearing here:

```
PS C:\> gwmi win32\_networkadapter -Credential (Get-Credential)

cmdlet Get-Credential at command pipeline position 1

Supply values for the following parameters:

Credential

gwmi : User credentials cannot be used for local connections
At line:1 char:
+ gwmi win32\_networkadapter -Credential (Get-Credential)
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  + CategoryInfo     : InvalidOperation: (:) [Get-WmiObject], ManagementException
  + FullyQualifiedErrorId : GetWMIManagementException,Microsoft.PowerShell.Comman

 ds.GetWmiObjectCommand
```

This error, for local connections, is not a Windows PowerShell thing, WMI has always behaved in this manner, even going back to the VBScript days.

Using the NetAdapter module

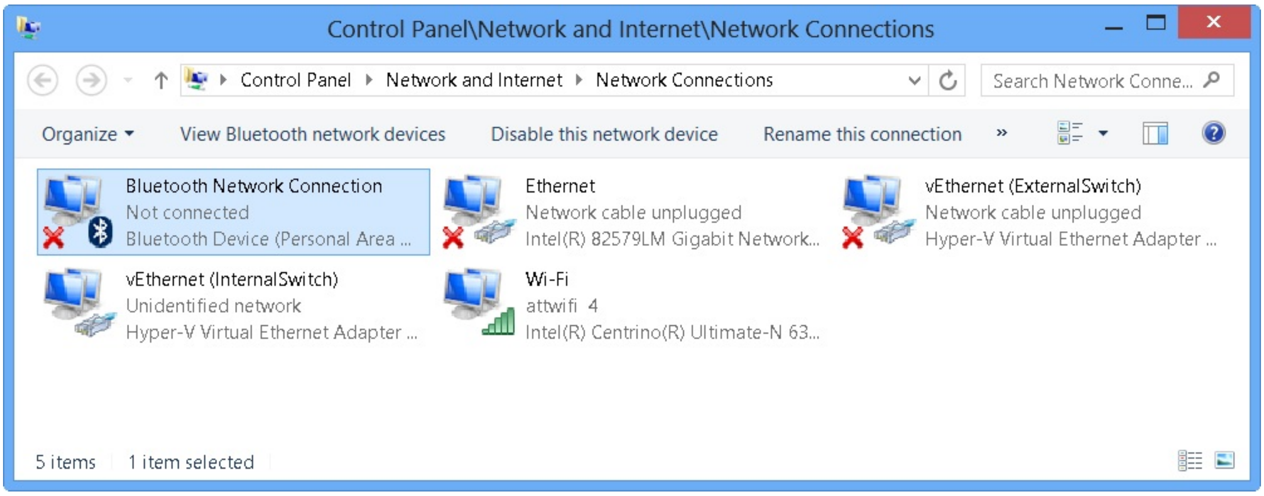
In Windows 8 (and above), I can use Windows PowerShell to stop or to start a network adapter by using one of the CIM commands. Of course, the function wraps the WMI class, but it also makes things really easy. The _netadapter _functions appear here (gcm is an alias for the Get-Command cmdlet)

```
PS C:\> gcm -Noun netadapter | select name, modulename

Name                    ModuleName
----                    ----------
Disable-NetAdapter             NetAdapter
Enable-NetAdapter             NetAdapter
Get-NetAdapter               NetAdapter
Rename-NetAdapter             NetAdapter
Restart-NetAdapter             NetAdapter
Set-NetAdapter               NetAdapter
```

NOTE: To enable or to disable network adapters requires admin rights. Therefore you must start the Windows PowerShell console with an account that has rights to perform the task.

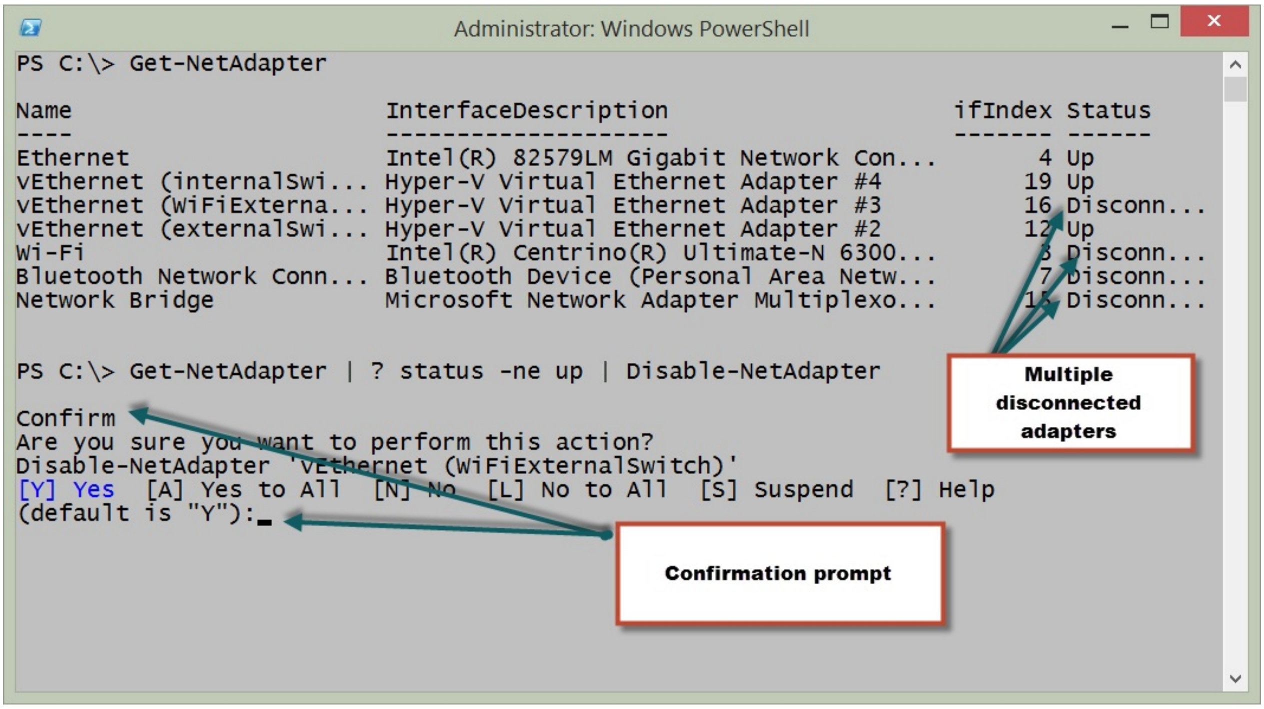
The various network adapters on my laptop appear in the figure that follows.



I do not like having enabled, disconnected network adapters. Instead, I prefer to only enable the network adapter I am using (there are a number of reasons for this such as simplified routing tables, performance issues, and security concerns). In the past, I wrote a script, now I only need to use a Windows PowerShell command. If I only want to disable the non-connected network adapters, the command is easy. It appears here.

```
Get-NetAdapter| ? status-ne up| Disable-NetAdapter
```

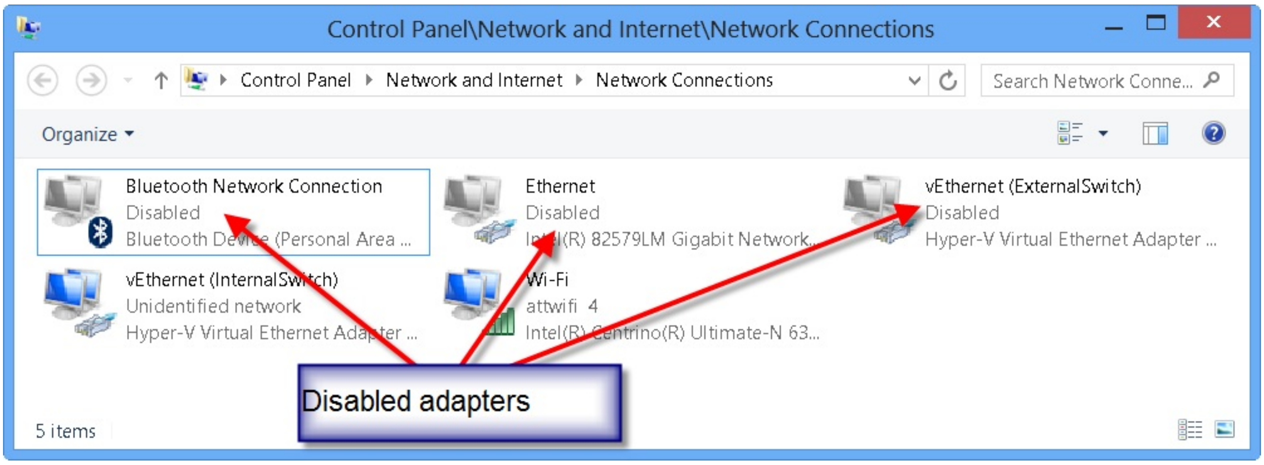
The problem with the previous command is that it prompts. This is not much fun when there are multiple network adapters to disable. The prompt appears here.



To suppress the prompt, I need to supply $false to the -confirm parameter. This appears here.

```
Get-NetAdapter| ? status-ne up | Disable-NetAdapter -Confirm:$false
```

A quick check in control panel shows the disconnected adapters are now disabled. This appears here.



If I want to enable a specific network adapter, I use the Enable-Network adapter. I can specify by name as appears here.

```
Enable-NetAdapter -Name ethernet -Confirm:$false
```

If I do not want to type the adapter name, I can use the Get-NetAdapter cmdlet to retrieve a specific network adapter and then enable it. This appears here.

```
Get-NetAdapter -Name vethernet\*| ? status -eq disabled| Enable-NetAdapter -Confirm:$fals
```

It is also possible to use wild card characters with the Get-NetAdapter to retrieve multiple adapters and pipeline them directly to the Disable-NetAdapter cmdlet. The following permits the confirmation prompts so that I can selectively enable or disable the adapter as I wish.

```
PS C:\> Get-NetAdapter -Name vethernet\* | Disable-NetAdapter

Confirm

Are you sure you want to perform this action?

Disable-NetAdapter 'vEthernet (InternalSwitch)'

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help

(default is "Y"):y

Confirm

Are you sure you want to perform this action?

Disable-NetAdapter 'vEthernet (ExternalSwitch)'

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help

(default is "Y"):n
```

Renaming the network adapter

Depending on the version of your operating system, you will have different capabilities available for renaming the network adapter. These methods involve using Netsh, WMI, and the functions from the NetAdapter module.

PowerTip : Renaming the network adapter

Question: You want to rename you network adapter. How can you do this using Windows PowerShell on Windows 8 or above?

Answer: Use the Get-Netadapter function to retrieve the specific network adapter and pipeline the results to the Rename-NetAdapter function. This technique appears here:

```
Get-NetAdapter -Name Ethernet|Rename-NetAdapter -NewName MyRenamedAdapter
```

Using NetSh

To rename the network adapter using NetSh I need to know the interface name, and the new name I want to use. This is about it. To find the network adapter names, I can also use NetSh. Here is the command:

```
netsh interface ipv4 show interfaces
```

NetSh is available everywhere right now. So, I can use NetSh to configure network adapters from Windows 2000 forward - so it has the greatest amount of backward compatibility. But it is deprecated, and therefore may not always be available going forward. To rename a network interface using NetSh, I can use a command such as the one appearing here:

```
NetSh interface set interface name="Ethernet" newname="RenamedAdapter"
```

Using WMI

Beginning with Windows Vista, it is possible to use WMI to rename the network interface. The thing to keep in mind, is that the property that I change is NetConnectionID and not the name property. Because this command modified the NetConnectionID property, it is a simple

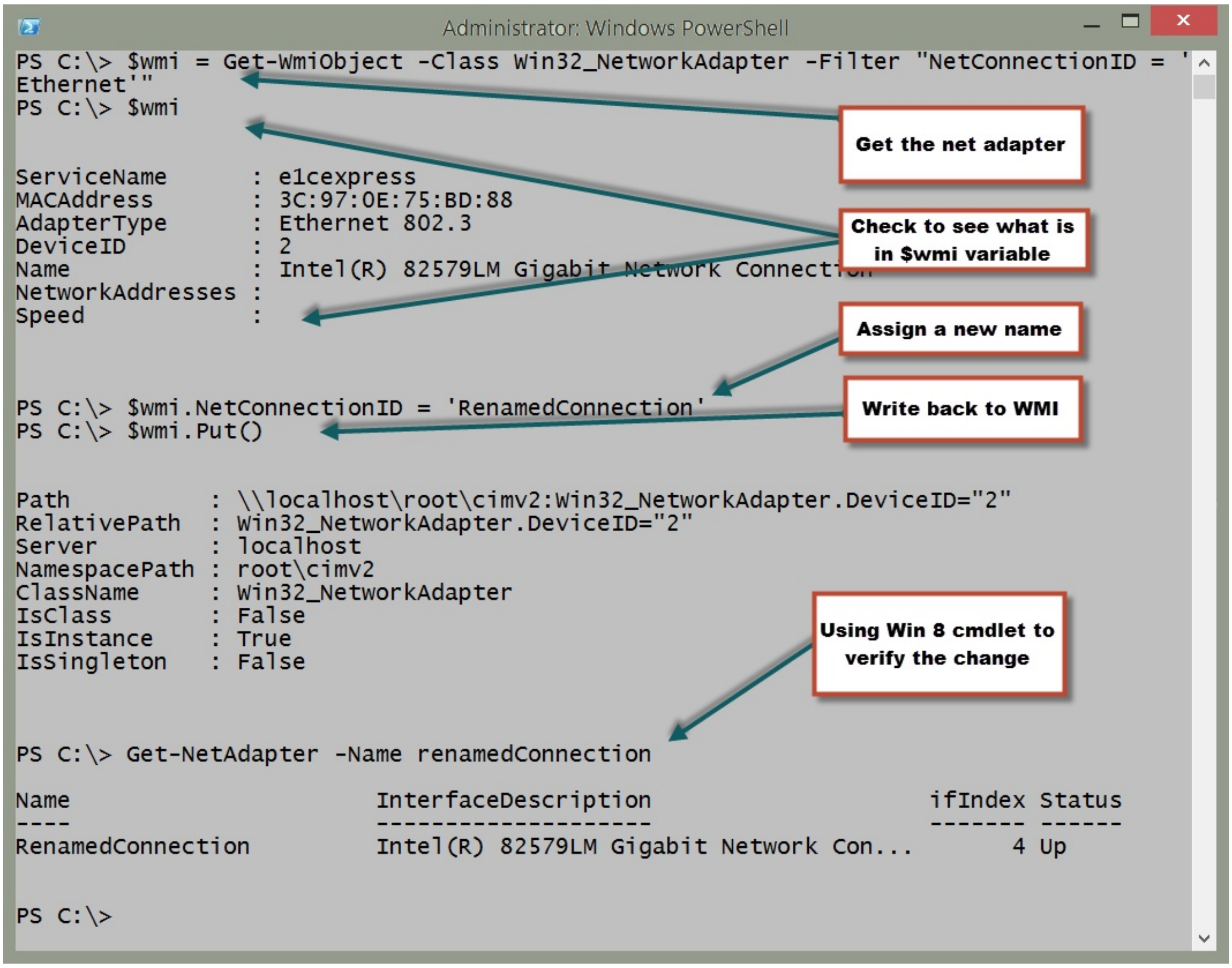
property assignment, and not a method call. The Win32_NetworkAdapter WMI class is documented on MSDN and the article shows the properties that are Read and Write. The steps to using WMI include the following:

1. Retrieve the specific instance of the network adapter
2. Assign a new value for the NetConnectionID property
3. Use the Put method to write the change back to WMI

The following code illustrates these three steps using a network adapter that is named Ethernet. The command will rename the network adapter named Ethernet to RenamedConnection:

```
$wmi = Get-WmiObject -Class Win32_NetworkAdapter -Filter "NetConnectionID = 'Ethernet'"
$wmi.NetConnectionID = 'RenamedConnection'
$wmi.Put()
```

The following figure shows using WMI to rename the network adapter.



Using WMI on Windows 7 and above

On Windows 7 and Windows Server 2008 R2, it is not necessary to use the Get-WmiObject cmdlet, assign new values for the property and call the Put method. This is because the Set-CimInstance cmdlet permits accomplishing this feat as single command. The easiest way to use Set-CimInstance is to use a query. Interestingly enough, this WQL query is the same type of query that would have been used back in the VBScript days. The query to retrieve the network adapter named Ethernet appears here:

```
"Select \* from Win32\_NetworkAdapter where NetConnectionID = 'EtherNet'"
```

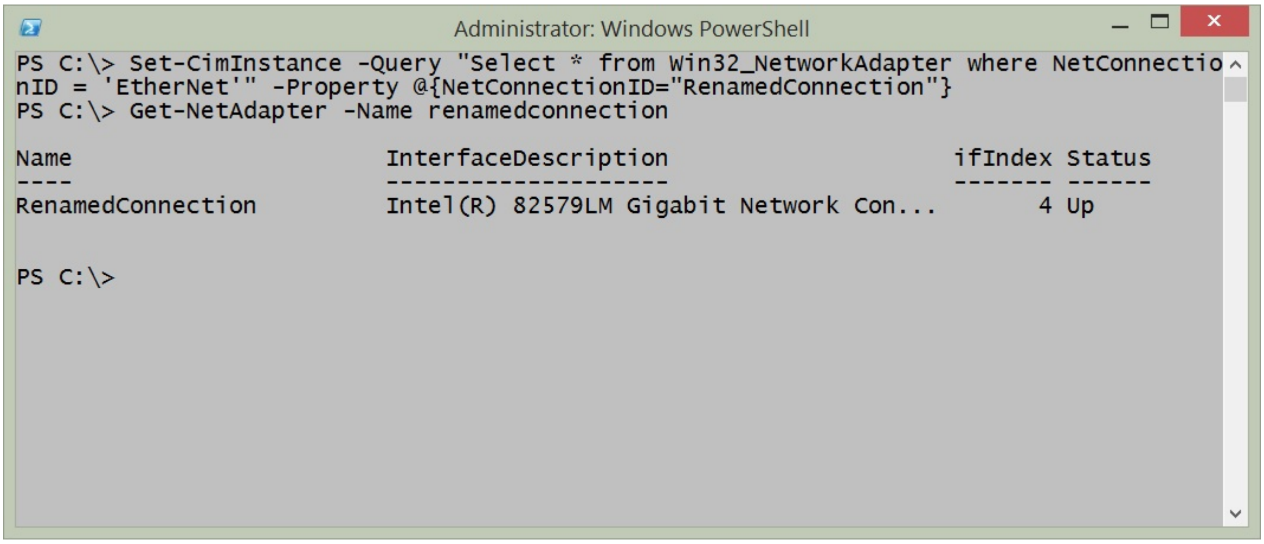
To assign a new value for a property, I use a hashtable. The hashtable specifies the property and the new value for the property. The hashtable to specify a value of RenamedConnection for the NetConnectionID property appears here:

```
@{NetConnectionID="RenamedConnection"}
```

The complete Set-CimInstance command appears here (this is a single line command)

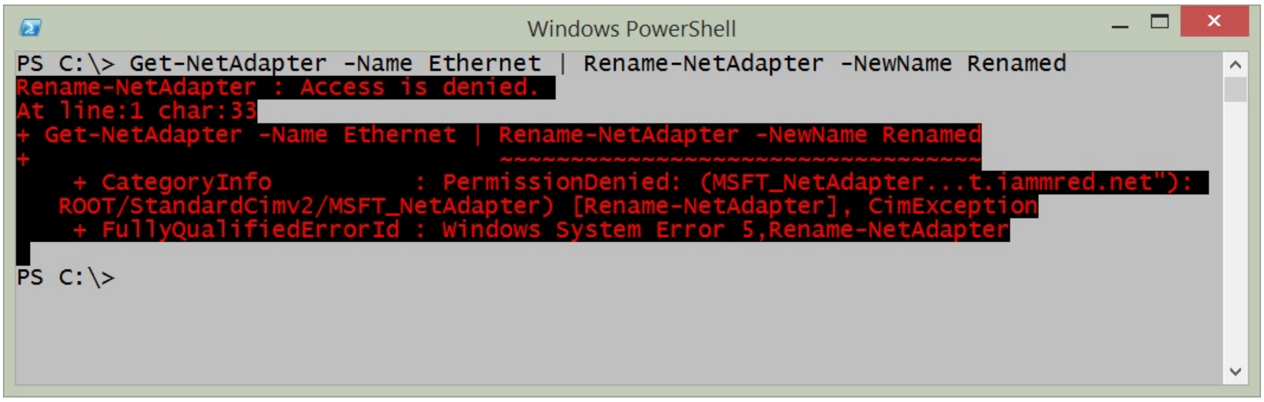
```
Set-CimInstance-Query"Select \* from Win32\_NetworkAdapter where NetConnectionID = 'Ether
```

When I run the command, nothing appears in the output. This following figure shows the single command (wrapping in the Windows PowerShell console) and the fact that there is not output from the command. On my Windows 8.1 laptop, I use the Get-NetAdapter command to verify that the adapter renamed.



Using the NetAdapter module

Renaming a network adapter via Windows PowerShell requires admin rights. Unfortunately, the help does not mention this. You just have to sort of know this. Luckily, an error occurs when attempting to run the command without admin rights. The error is instructive, and informs that access is denied. The error appears here.



The good thing is that the _access denied _error appears - some cmdlets do not display output, and do not let you know that you need admin rights to obtain the information (The Get-VM cmdlet is one of those. It returns no virtual machine information, but it does not generate an error either. This situation is also true with the Start-VM cmdlet -- does not do anything, but does not generate an error if you do not have rights).

So I close the Windows PowerShell console, right click on the Windows PowerShell console icon I created on my task bar, and run Windows PowerShell as Administrator. I now run the command to rename my network adapter with the _whatif _parameter to ensure it accomplishes what I want. Here is the command I use:

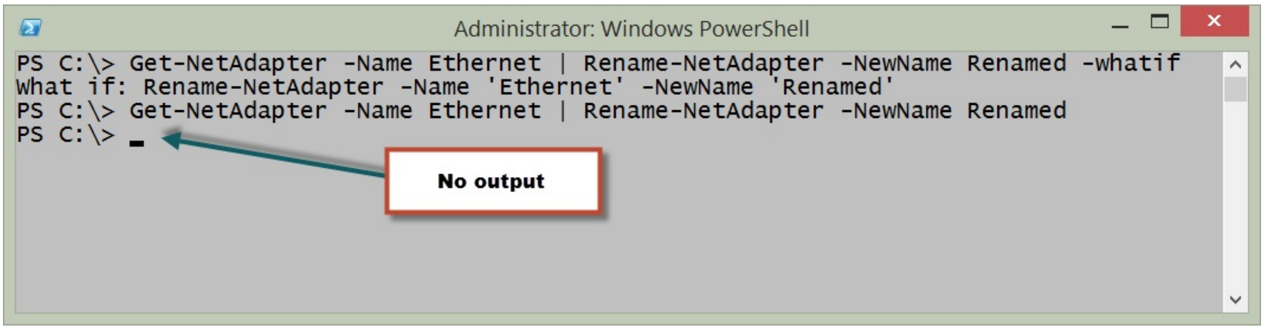
```
Get-NetAdapter -Name Ethernet | Rename-NetAdapter -NewName Renamed -whatif

What if: Rename-NetAdapter -Name 'Ethernet' -NewName 'Renamed'
```

That is exactly what I want to happen. I now use the up arrow, and remove the _whatif. _Here is the command (no output returns from this command).

```
Get-NetAdapter -Name Ethernet | Rename-NetAdapter -NewName Renamed
```

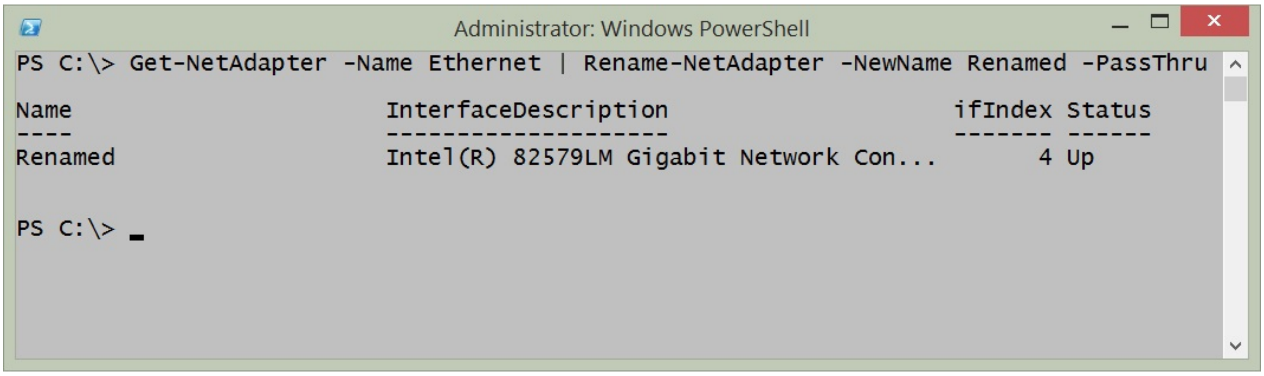
The command, and associated output appear in the figure here.



I can modify my command just a bit, and return an instance of the renamed network adapter. To do this, I use the -passthru parameter from the Rename-NetAdapter function. One reason to do this is to see visual confirmation that the command completed successfully. Other reasons, would be to use the returned object to feed into other cmdlets and to perform other actions. Here is the revised command, showing how to use -passthru

```
Get-NetAdapter -Name Ethernet | Rename-NetAdapter -NewName Renamed –PassThru
```

The command, and associated output appear in the figure that follows.



One of the really powerful things about the Get-NetAdapter function is that I can use wildcard characters for the name parameter. This means that if I do not want to type the entire network adapter name, I can shorten it. It also means that if I have a similar naming pattern, I can use a wildcard pattern to retrieve them as well. Here is an example of using a wildcard.

```
Get-NetAdapter -NameEther\*
```

This command works the same as the other commands, and therefore I can pipeline the results to the Rename-NetAdapter function. This technique appears here:

```
Get-NetAdapter -Name Ether\* | Rename-NetAdapter -NewName Renamed -PassThru
```

As seen in the figure here, the command works perfectly.

```
Administrator: Windows PowerShell
PS C:\> Get-NetAdapter -Name Ether* | Rename-NetAdapter -NewName Renamed -PassThru

Name                      InterfaceDescription                    ifIndex Status
----                      --------------------                    ------- ------
Renamed                   Intel(R) 82579LM Gigabit Network Con...       4 Up


PS C:\> _
```

Finding connected network adapters

One of the most fundamental pieces of troubleshooting or security checks to do is to find out which of the many network adapters on a computer are actually connected to a network.

PowerTip : Show 'up' physical adapters

Question: You want to see which physical network adapters on your Windows 8.1 computer using Windows PowerShell. How can you do this?

Answer: Use the -physical parameter with the Get-NetAdapter function and filter for a status of up. This technique appears here:

```
Get-NetAdapter -physical| where status -eq 'up'
```

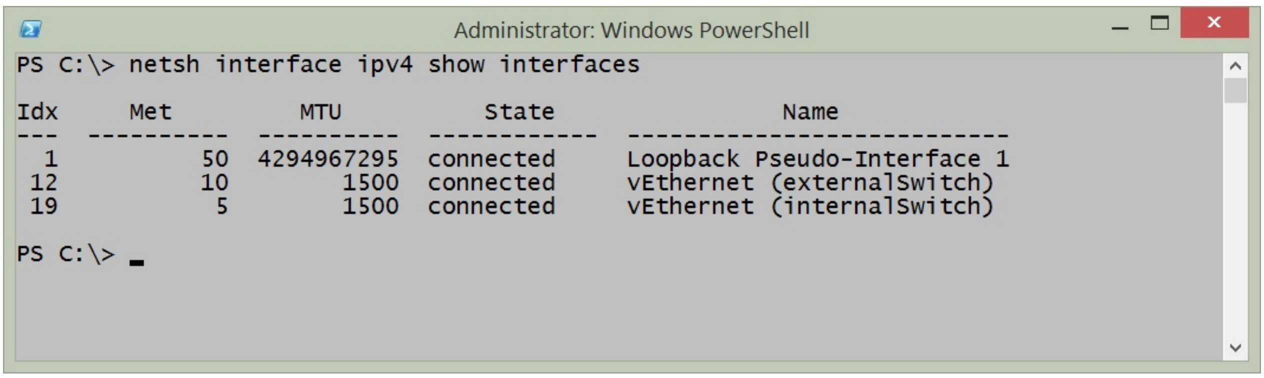
Using NetSh

It is pretty easy to use NetSh to retrieve information about the connection status of network adapters. To do so, I use the following command:

```
netsh interface ipv4 show interfaces
```

One of the problems, from a management perspective, is that the command returns text. Therefore, if I need to parse the text to pull out specific information, such as the Interface Index number, or the Name of the adapter, then I am going to have to resort to writing a complicated regular expression pattern. If all I need to do is to obtain the information because I am writing to a log file as text, then the command works great, and is the lowest common denominator - I can use it all the way back to Windows 2000 days.

I can even run the netsh commands from within the Windows PowerShell console. This appears in the figure that follows.



Using WMI

It is possible to use WMI and the Win32_NetworkAdapter WMI class to retrieve information about the connection status. The NetConnectionStatus property reports backed in a coded value that reports the status. These values are documented on MSDN for the Win32_NetworkAdapter class. Using the Get-WmiObject Windows PowerShell cmdlet, I can work with any operating system that installs Windows PowerShell. This includes Windows XP, Windows Server 2003 and above. The following command returns information similar to the NetSh command.

```
get-wmiobject win32\_networkadapter|select netconnectionid, name, InterfaceIndex, netconn
```

The command and the output from the command appear in the figure that follows.

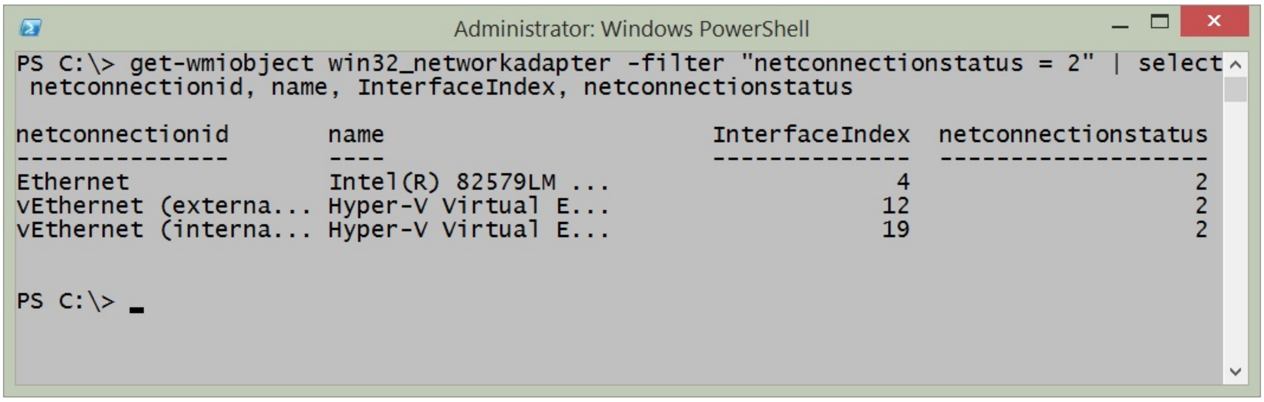
```
![image045.png](images/image045.png)
```

The difference is that instead of plain text, the command returns objects that can be further manipulated. Therefore, while the above command actually returns the network connection status of all network adapters, the NetSh command only returns the ones that are connected. If I filter on a netconnectionstatus of 2 I can return only the connected network adapters. The command becomes this one (this is a single line command that I broke at the pipeline character for readability):

```
get-wmiobject win32\_networkadapter -filter "netconnectionstatus = 2"|

selectnetconnectionid,name,InterfaceIndex,netconnectionstatus
```

The command and output appear in the figure that follows.



If the desire is to obtain the connection status of more than just network adapters that are connected, then the task will require writing a script to do a lookup. The lookup values appear in the table that follows:

Value	Meaning
0	Disconnected
1	Connecting
2	Connected
3	Disconnecting
4	Hardware not present
5	Hardware disabled
6	Hardware malfunction
7	Media disconnected
8	Authenticating
9	Authentication succeeded
10	Authentication failed
11	Invalid Address
12	Credentials required

The Get-NetworkAdapterStatus.ps1 script requires at least Windows PowerShell 2.0 which means that it will run on Windows XP SP3 and above.

Get-NetworkAdapterStatus.Ps1

```
<#

 .Synopsis
  Produces a listing of network adapters and status on a local or remote machine.

 .Description
  This script produces a listing of network adapters and status on a local or remote mach

 .Example
  Get-NetworkAdapterStatus.ps1 -computer MunichServer
  Lists all the network adapters and status on a computer named MunichServer

 .Example
  Get-NetworkAdapterStatus.ps1
  Lists all the network adapters and status on local computer

 .Inputs
  [string]
```

```
 .OutPuts
  [string]

 .Notes
  NAME: Get-NetworkAdapterStatus.ps1

  AUTHOR: Ed Wilson

  LASTEDIT: 1/10/2014

  KEYWORDS: Hardware, Network Adapter

 .Link

  Http://www.ScriptingGuys.com

#Requires -Version 2.0

#>

Param(
 [string]$computer=$env:COMPUTERNAME
) #end param

functionGet-StatusFromValue
{
Param($SV)
switch($SV)
 {
 0 { " Disconnected" }
 1 { " Connecting" }
 2 { " Connected" }
 3 { " Disconnecting" }
 4 { " Hardware not present" }
 5 { " Hardware disabled" }
 6 { " Hardware malfunction" }
 7 { " Media disconnected" }
 8 { " Authenticating" }
 9 { " Authentication succeeded" }
 10 { " Authentication failed" }
 11 { " Invalid Address" }
 12 { " Credentials Required" }
 Default { "Not connected" }
 }

} #end Get-StatusFromValue function

# \*\*\* Entry point to script \*\*\*

Get-WmiObject-Classwin32\_networkadapter-computer$computer|
Select-ObjectName, @{LABEL="Status";
```

```
EXPRESSION={Get-StatusFromValue$\_.NetConnectionStatus}}
```

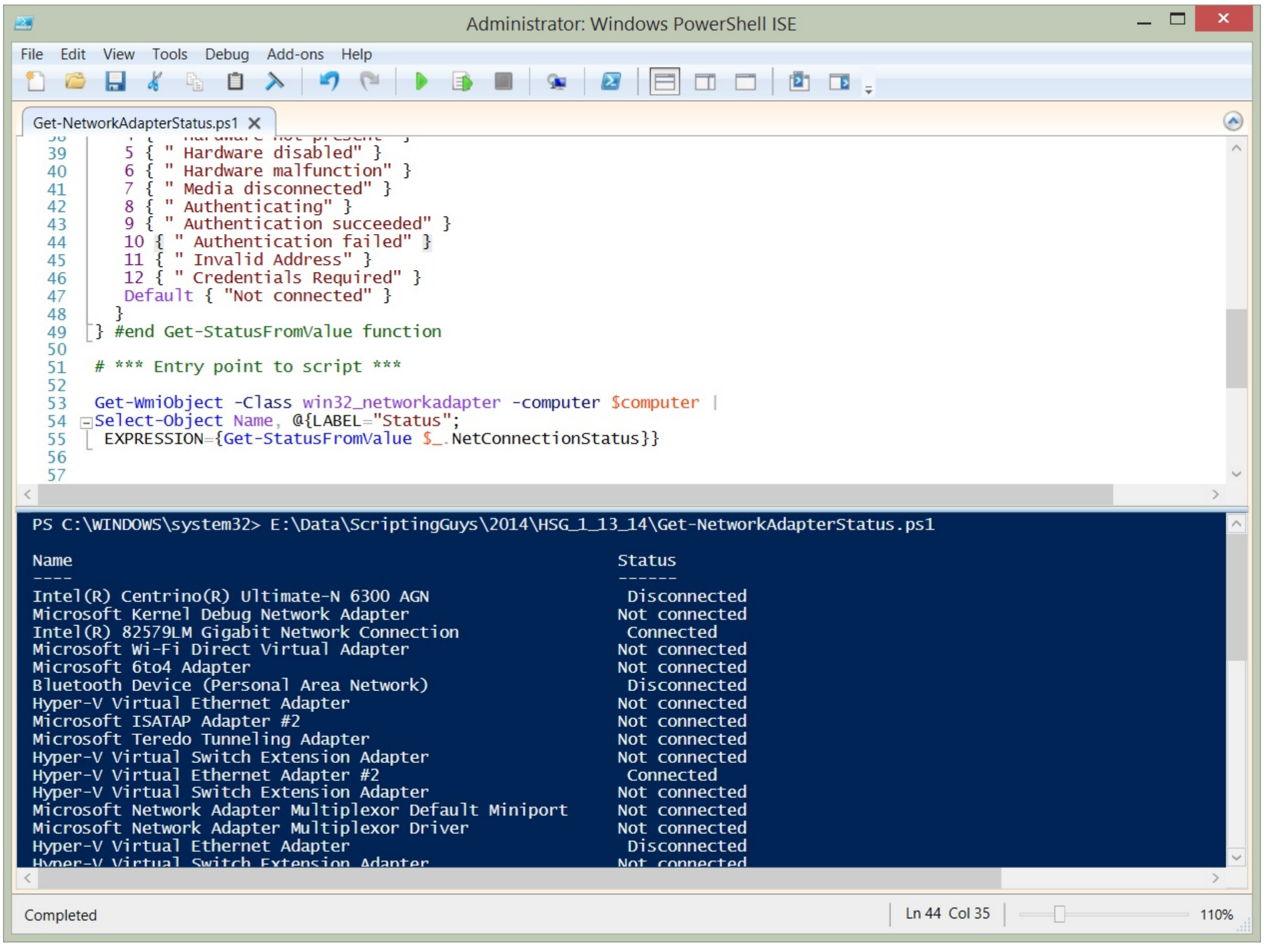
If my environment is Windows 7 and Windows Server 2008 R2, I can use either Windows PowerShell 3.0 or Windows PowerShell 4.0. The advantage here, is that I gain access to the Get-CimInstance cmdlet which uses WinRM for remoting instead of DCOM that the Get-WmiObject cmdlet uses. The only change to the Get-NetworkAdapterStatus.ps1 script that is required is to replace the Get-WmiObject line with Get-CimInstance. The revision appears here:

```
# \*\*\* Entry point to script \*\*\*

Get-CimInstance-Classwin32\_networkadapter-computer$computer|
Select-ObjectName, @{LABEL="Status";

EXPRESSION={Get-StatusFromValue$\_.NetConnectionStatus}}
```

When I run the Get-StatusFromValue.ps1 script, in the Windows PowerShell ISE, I see the output achieved here.

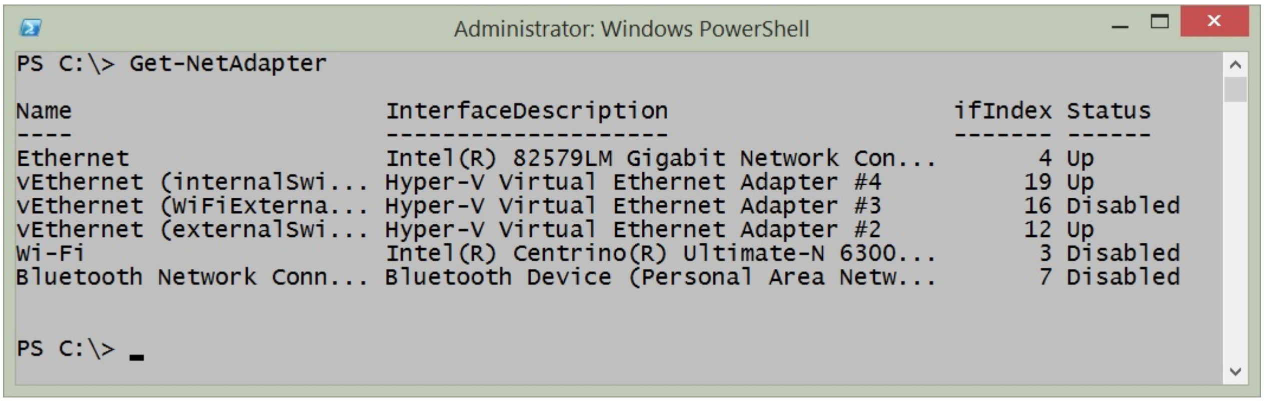


Using the NetAdapter module

On Windows 8 and above the NetAdapter module contains the Get-NetAdapter function. To see the status of all network adapters, use the Get-NetAdapter function with no parameters. The command appears here:

```
Get-NetAdapter
```

The output from this command appears here.



```
Administrator: Windows PowerShell
PS C:\> Get-NetAdapter

Name                      InterfaceDescription                    ifIndex Status
----                      --------------------                    ------- ------
Ethernet                  Intel(R) 82579LM Gigabit Network Con...       4 Up
vEthernet (internalSwi... Hyper-V Virtual Ethernet Adapter #4          19 Up
vEthernet (WiFiExterna... Hyper-V Virtual Ethernet Adapter #3          16 Disabled
vEthernet (externalSwi... Hyper-V Virtual Ethernet Adapter #2          12 Up
Wi-Fi                     Intel(R) Centrino(R) Ultimate-N 6300...       3 Disabled
Bluetooth Network Conn... Bluetooth Device (Personal Area Netw...       7 Disabled


PS C:\> _
```

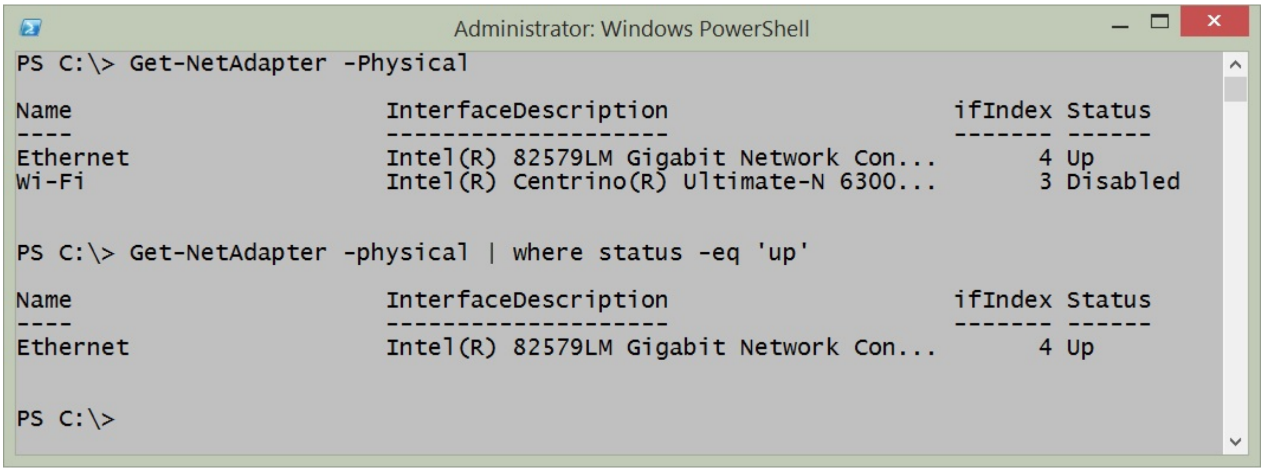
I can reduce the output to only physical adapters by using the -physical parameter. This command appears here.

```
Get-NetAdapter-Physical
```

If I only want to see the physical network adapters that are actually up, I pipeline the results to the where-object. This command appears here.

```
Get-NetAdapter-physical|wherestatus-eq'up'
```

The commands and the output from the two previous commands appear in the figure that follows.



```
Administrator: Windows PowerShell
PS C:\> Get-NetAdapter -Physical

Name                      InterfaceDescription                    ifIndex Status
----                      --------------------                    ------- ------
Ethernet                  Intel(R) 82579LM Gigabit Network Con...       4 Up
Wi-Fi                     Intel(R) Centrino(R) Ultimate-N 6300...       3 Disabled


PS C:\> Get-NetAdapter -physical | where status -eq 'up'

Name                      InterfaceDescription                    ifIndex Status
----                      --------------------                    ------- ------
Ethernet                  Intel(R) 82579LM Gigabit Network Con...       4 Up

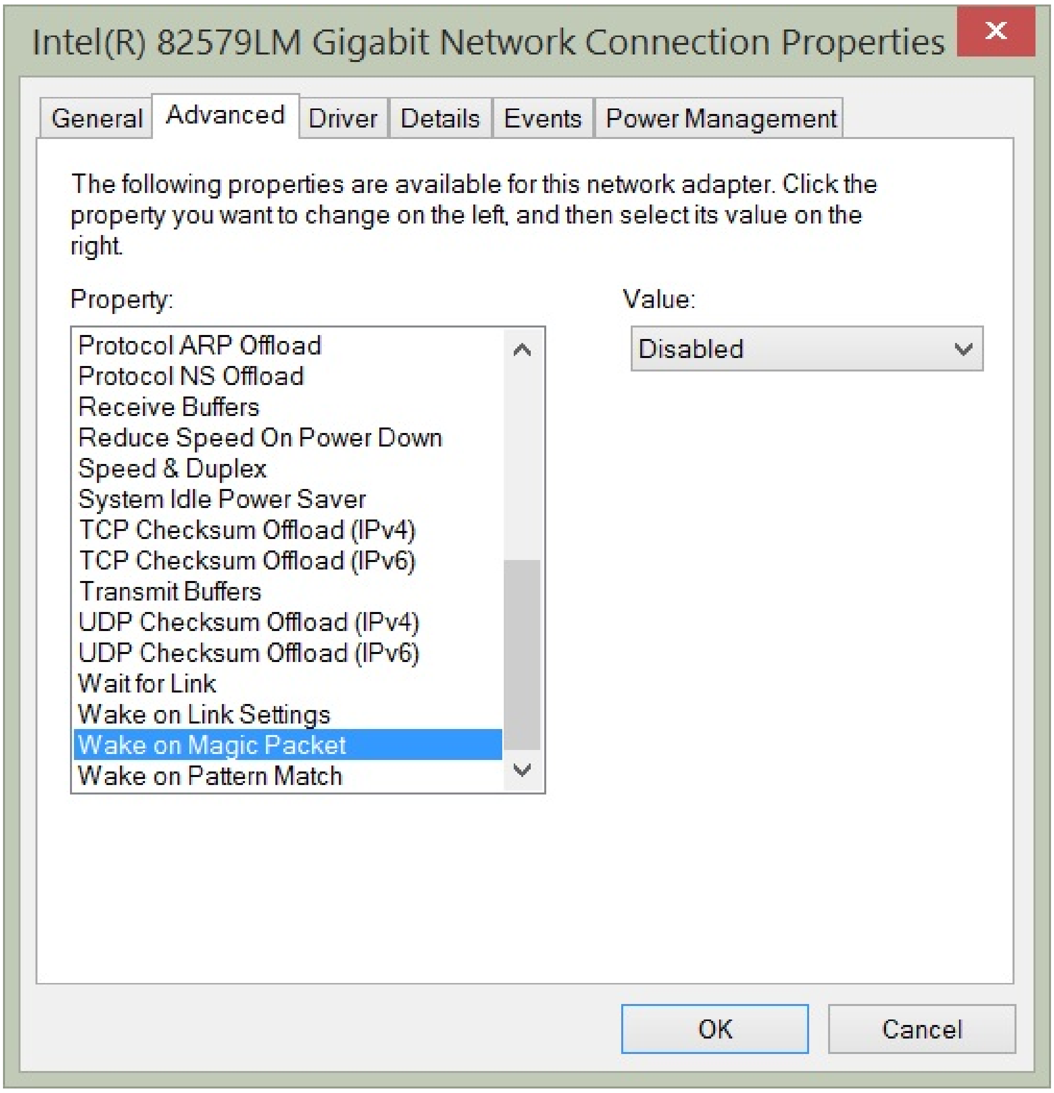

PS C:\>
```

Network Adapter power settings

Beginning with Windows 7, the network adapter power settings expanded.aspx). Some of the improvements are listed here:

- Wake on LAN and Wake on Wireless LAN. Improved wake patterns reduce the number of false wakes. Beginning with Windows 7 a directed packet (such as a ping) does not cause the computer to wake up.
- ARP (Address Resolution Protocol) and NS (Neighbor Solicitation) offload. ARP and NS packets do not wake up the computer. Instead the network adapter, beginning with Windows 7, can respond. Therefore the computer does not need to wake up just to maintain a presence on the network. This support depends on at least NDIS 6.0 drivers and may not be available with older hardware.
- Low Power on Media Disconnect. Enables the computer to place the network adapter into a low power state when the network cable is unplugged and the computer is running.

These settings are configurable via the graphical user interface by selecting the configure button from the network adapter properties dialog box. The settings appear in the figure that follows.



PowerTip : Get network adapter power management settings

Question: You want to get the network adapter power management settings on your Window 8.1 computer. How can you use Windows PowerShell to do this?

Answer: Use the Get-NetAdapterPowerManagement function and specify the name of the network adapter to query.

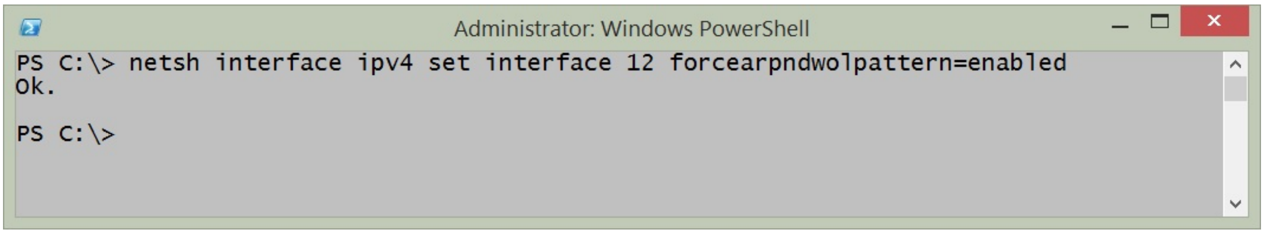
```
Get-NetAdapterPowerManagement-Name ethernet
```

Using NetSh

Some of the network adapter power management settings are configurable via NetSh. For example, to permit ARP packets and NS packets to wake the network adapter, I would use a command such as the following:

```
netsh interface ipv4 set interface 12 forcearpndwolpattern=enabled
```

When the command completes successfully, it returns OK. Keep in mind, this will also cause a network adapter reset. The command and associated output appear here:

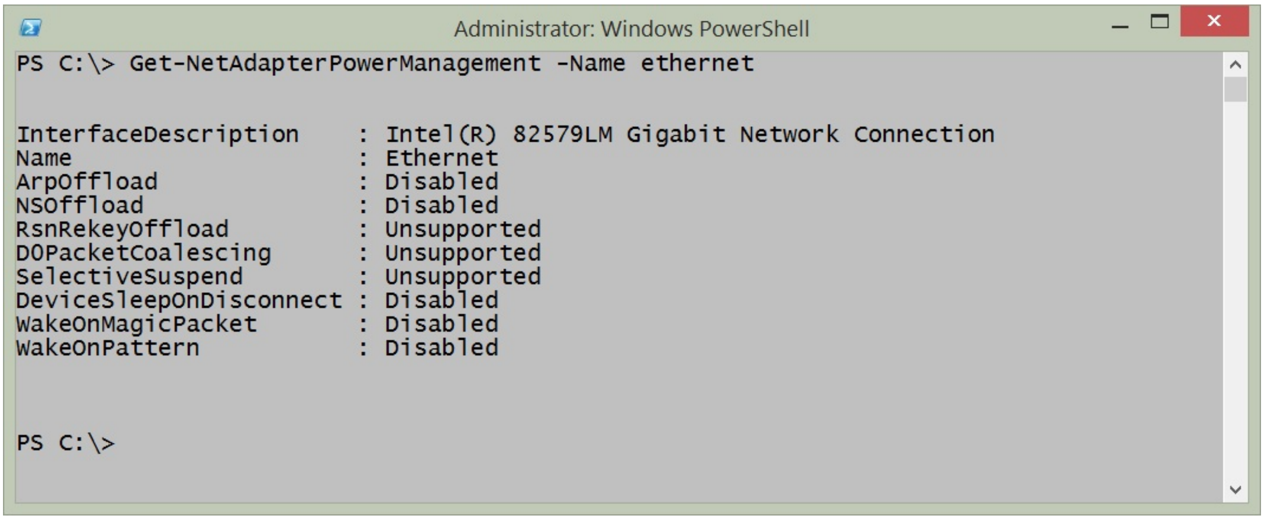


Using the NetAdapter module

To query the power management settings for a specific network adapter, use the Get-NetAdapterPowerManagement function and specify the name of the network adapter. An example of the command appears here:

```
Get-NetAdapterPowerManagement -Name ethernet
```

The command, and the output associated with the command appear in the figure that follows.



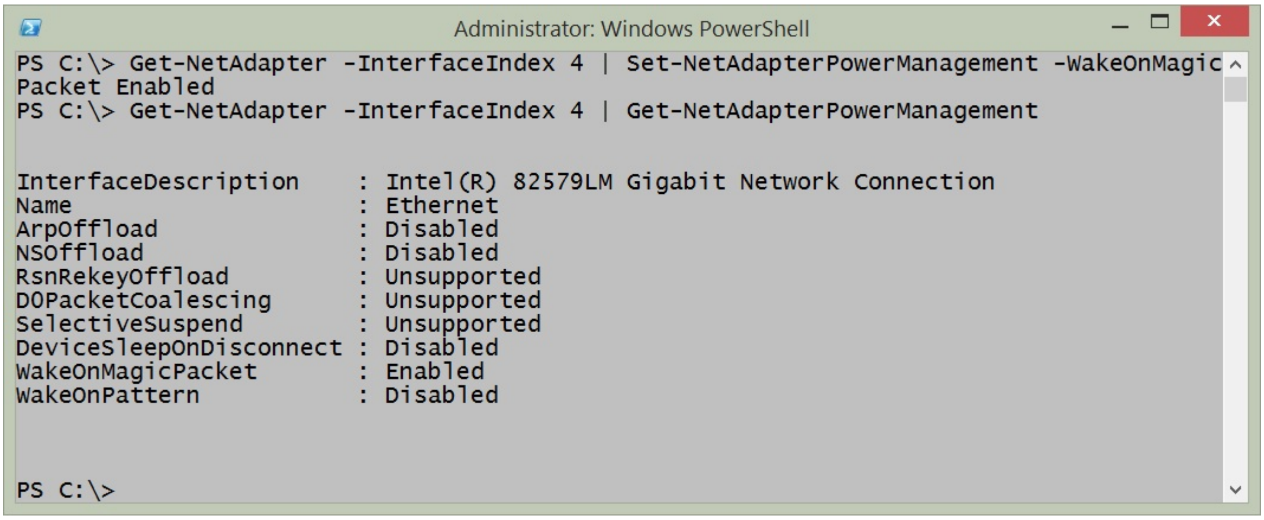
The Get-NetAdapterPowerManagement function only permits the use of the adapter name or interface description as parameters. But the Get-NetAdapter function is much more flexible. I often use Get-NetAdapter to retrieve a specific network adapter and then pipeline it to other functions such as Get-NetAdapterPowerManagement. This technique appears here:

```
Get-NetAdapter -Interface Index4 | Get-NetAdapterPowerManagement
```

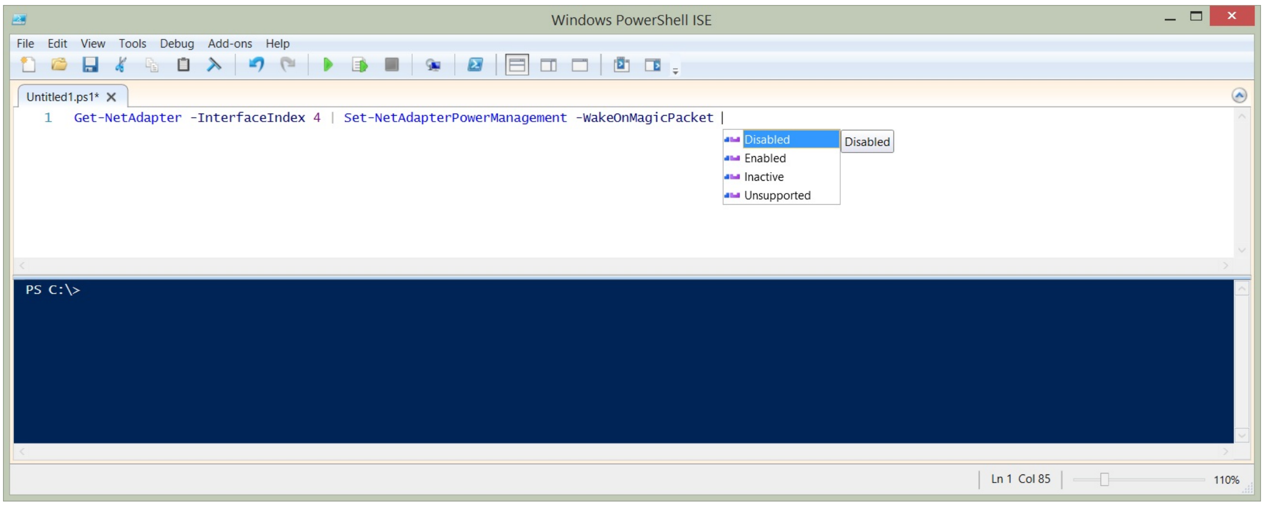
To configure the network adapter power management, I use the Set-NetAdapterPowerManagement function. Once again, I want to retrieve my network adapter by interface index number instead of having to type the name or description of the adapter. I pipeline the resulting network adapter object to the Set-NetAdapterPowerManagement function and specify a value for the -WakeOnMagicPacket parameter. The command appears here.

```
Get-NetAdapter -Interface Index4 | Set-NetAdapterPowerManagement -WakeOnMagicPacket Enabl
```

Because no output returns from the command, I use the Get-NetAdapter command a second time to verify the configuration change took place. The commands and associated output appear in the figure that follows.



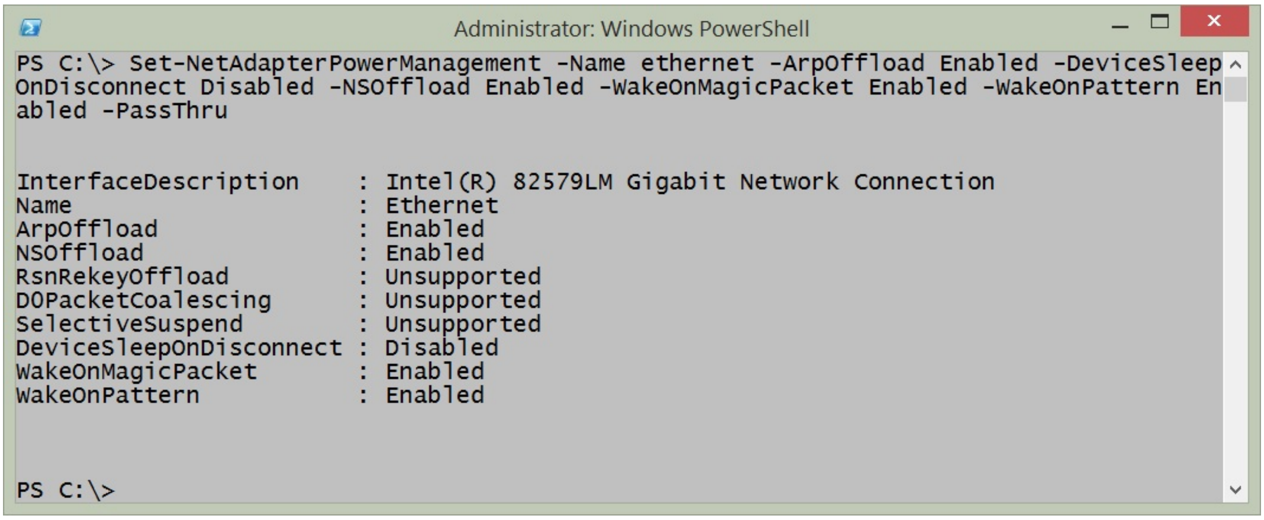
Now, it so happens that I know what the permissible values are for the parameters. But if I did not know this, I could create the command in the Windows PowerShell ISE and rely upon the intellisense features. When I type a parameter name, the permissible values appear and make it possible to select the correct value from the list. This appears in the figure that follows.



Most of the times, when I need to manage network adapter power management settings, it is because of a new deployment, or because an audit has determined that I have configuration drift. (Hmmm - this would actually be a great thing to use Desired Configuration Management to control.) So, what I do is put all the settings I want to configure into a single command. Such a command appears here:

```
Set-NetAdapterPowerManagement -Name ethernet -ArpOffload Enabled -DeviceSleep OnDisconnec
```

The -passthru parameter outputs a configuration management object so that I can inspect it and ensure that the proper things change that I wanted changed. The command, and the output from the command appear in the figure that follows:



```
Administrator: Windows PowerShell
PS C:\> Set-NetAdapterPowerManagement -Name ethernet -ArpOffload Enabled -DeviceSleep
OnDisconnect Disabled -NSOffload Enabled -WakeOnMagicPacket Enabled -WakeOnPattern En
abled -PassThru


InterfaceDescription       : Intel(R) 82579LM Gigabit Network Connection
Name                       : Ethernet
ArpOffload                 : Enabled
NSOffload                  : Enabled
RsnRekeyOffload            : Unsupported
D0PacketCoalescing         : Unsupported
SelectiveSuspend           : Unsupported
DeviceSleepOnDisconnect    : Disabled
WakeOnMagicPacket          : Enabled
WakeOnPattern              : Enabled


PS C:\>
```

To make changes to multiple computers, I first use the New-CimSession cmdlet to make my remote connections. I can specify the computer names and the credentials to use to make the connection. I then store the remote connection in a variable. Next, I pass that cimsession to the -cimsession parameter. The key to remember here, is that I must be able to identify the network adapter that I need to use for the management activity. An example of creating a Cim Session and using it appears here (keep in mind this is a two line command. If you directly copy and paste this command you must change the computer name, network interface name, and remove spaces until the second command appears on a single line).

```
$session = New-CimSession -ComputerName edlt
Set-NetAdapterPowerManagement -CimSession $session -name ethernet -ArpOffload Enabled -De
```

The command and the output from the command appear in the figure that follows.

```
Administrator: Windows PowerShell
PS C:\> $session = New-CimSession -ComputerName edlt
PS C:\> Set-NetAdapterPowerManagement -CimSession $session -name ethernet -ArpOffload
 Enabled -DeviceSleepOnDisconnect Disabled -NSOffload Enabled -WakeOnMagicPacket Enab
led -WakeOnPattern Enabled -PassThru


InterfaceDescription    : Intel(R) 82579LM Gigabit Network Connection
Name                    : Ethernet
ArpOffload              : Enabled
NSOffload               : Enabled
RsnRekeyOffload         : Unsupported
D0PacketCoalescing      : Unsupported
SelectiveSuspend        : Unsupported
DeviceSleepOnDisconnect : Disabled
WakeOnMagicPacket       : Enabled
WakeOnPattern           : Enabled
PSComputerName          : edlt



PS C:\>
```

Keep in mind that these commands require that either the Windows PowerShell console or the Windows PowerShell ISE is opened with admin rights. To do this, right click on the Windows PowerShell console icon or Windows PowerShell ISE icon while holding down the shift key and select run as administrator. Or if you launch it via Windows Search on Windows 8.1 type Windows PowerShell from the Start page, and the Search dialog appears with the Windows PowerShell icon. Right click on the icon and select Run as Administrator from the action menu.

If you do not launch Windows PowerShell with admin rights, an error occurs stating that it cannot find the network adapter. An example of the error appears in the figure that follows.

```
Windows PowerShell
PS C:\> Get-NetAdapterPowerManagement -Name ethernet
Get-NetAdapterPowerManagement : No MSFT_NetAdapterPowerManagementSettingData
objects found with property 'Name' equal to 'ethernet'.  Verify the value of the
property and retry.
At line:1 char:1
+ Get-NetAdapterPowerManagement -Name ethernet
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (ethernet:String) [Get-NetAdapterPowe
   rManagement], CimJobException
    + FullyQualifiedErrorId : CmdletizationQuery_NotFound_Name,Get-NetAdapterPowerM
   anagement

PS C:\>
```

Getting Network Statistics

One of the cool things about the Windows platform are all the different ways of obtaining networking statistical information. There are things like NetStat, NetSh, performance counters, as well as the Get-NetworkStatistics function from the NetAdapter Windows PowerShell module. All of these methods can be used inside the Windows PowerShell console, or from within the Windows PowerShell ISE.

PowerTip : Use PowerShell to find Networking counters

Question: You need to check on the network performance, but do not know where to begin. How can you use Windows PowerShell to find networking counters?

Answer: Use the Get-Counter cmdlet and the -ListSet parameter. Select the CounterSetName property and filter on names related to networking. The following command returns sets related to IPV6.

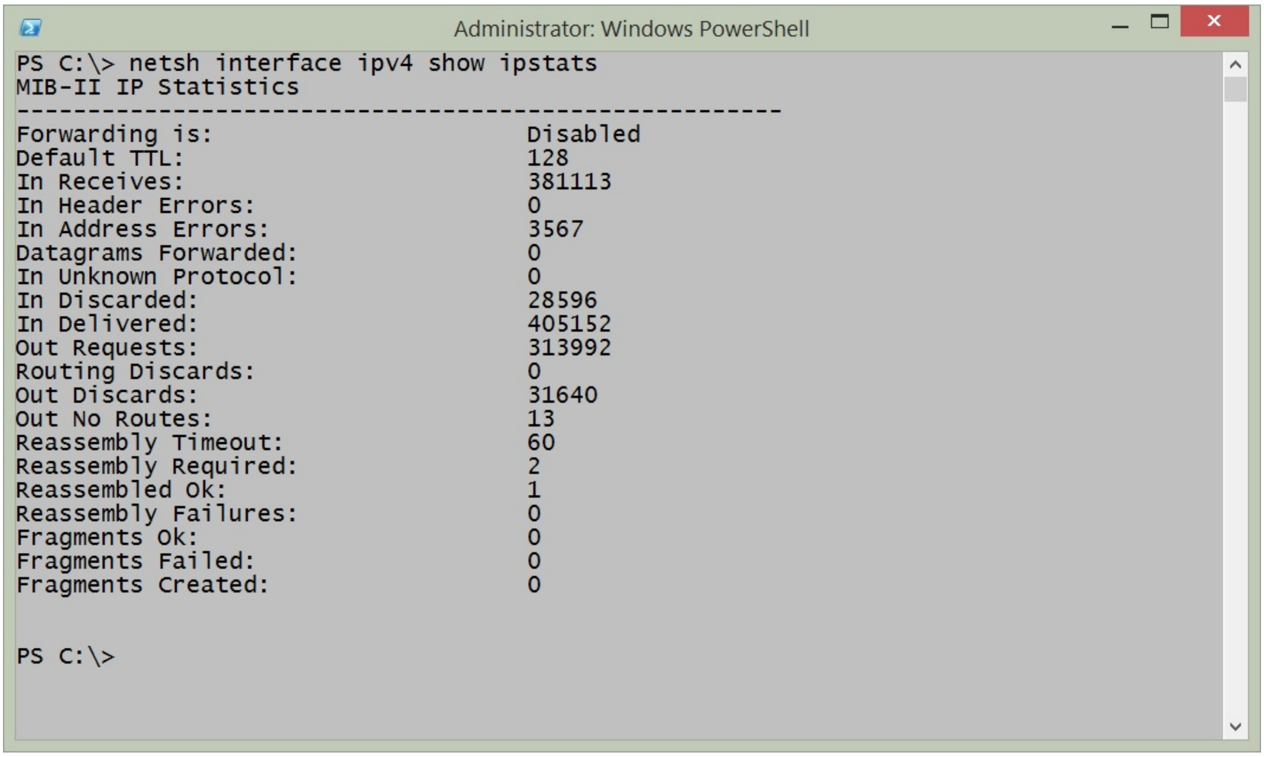
```
Get-Counter -ListSet * | select countersetname | where countersetname -match 'ipv6'
```

NetSh

Using NetSh to obtain network statistics is easy and powerful. For example to show IP statistics, I use the command appearing here.

```
netsh interface ipv4 show ipstats
```

A sample output from this command appears in the figure that follows.



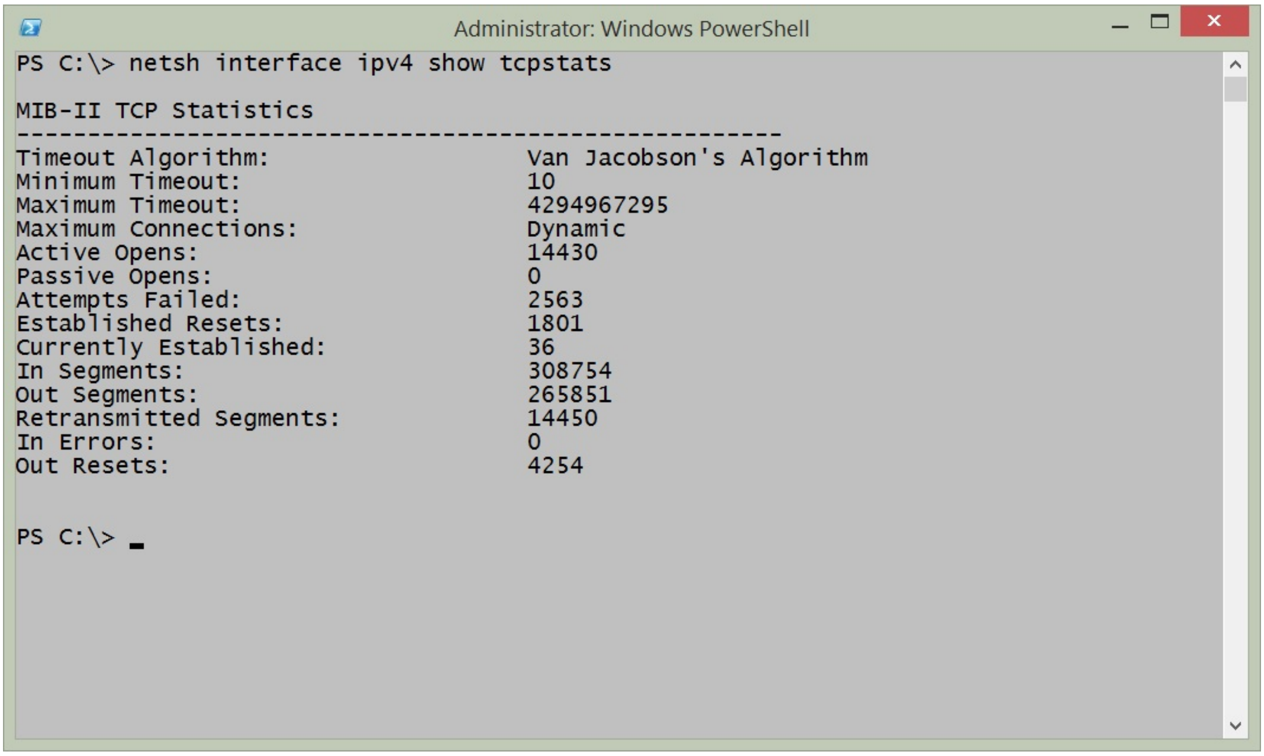
```
Administrator: Windows PowerShell
PS C:\> netsh interface ipv4 show ipstats
MIB-II IP Statistics
------------------------------------------------------
Forwarding is:                      Disabled
Default TTL:                        128
In Receives:                        381113
In Header Errors:                   0
In Address Errors:                  3567
Datagrams Forwarded:                0
In Unknown Protocol:                0
In Discarded:                       28596
In Delivered:                       405152
Out Requests:                       313992
Routing Discards:                   0
Out Discards:                       31640
Out No Routes:                      13
Reassembly Timeout:                 60
Reassembly Required:                2
Reassembled Ok:                     1
Reassembly Failures:                0
Fragments Ok:                       0
Fragments Failed:                   0
Fragments Created:                  0


PS C:\>
```

To show TCP statistics using NetSh, I use the command appearing here.

```
netsh interface ipv4 show tcpstats
```

The command, and the output from the command appear in the figure that follows.



```
Administrator: Windows PowerShell
PS C:\> netsh interface ipv4 show tcpstats

MIB-II TCP Statistics
------------------------------------------------------
Timeout Algorithm:                  Van Jacobson's Algorithm
Minimum Timeout:                    10
Maximum Timeout:                    4294967295
Maximum Connections:                Dynamic
Active Opens:                       14430
Passive Opens:                      0
Attempts Failed:                    2563
Established Resets:                 1801
Currently Established:              36
In Segments:                        308754
Out Segments:                       265851
Retransmitted Segments:             14450
In Errors:                          0
Out Resets:                         4254


PS C:\> _
```

One of the cool things about using NetSh from within Windows PowerShell is that I have the power of Windows PowerShell at my fingertips. Rather than keep going back and forth to find stuff, I can pipeline the results from a command to the Select-String cmdlet. For example, if I am interested in how many commands are available to show statistics, I use the command appearing here because I noticed that each of the commands contains the letters stats:

```
netsh interface ipv4 show | Select-String "stats"
```

The output from the command appears here:

```
PS C:\> netsh interface ipv4 show | Select-String "stats"

show icmpstats - Displays ICMP statistics.

show ipstats  - Displays IP statistics.

show tcpstats - Displays TCP statistics.

show udpstats - Displays UDP statistics.
```

In addition to the IPV4 interface, I can also work with the IPV6 interface and obtain similar statistics. Here is the command I used to obtain that information:

```
PS C:\> netsh interface ipv6 show | Select-String "stats"

show ipstats  - Displays IP statistics.

show tcpstats - Displays TCP statistics.

show udpstats - Displays UDP statistics.
```

In addition to using the Select-String cmdlet to parse the output from the NetSh help, I can also use it to hone in on specific information from the statistics. For example, the following command retrieves IPv6 interface IP stats.

```
netsh interface ipv6 show ipstats
```

I can hone in on the output and look for errors by piplining the results to the Select-String cmdlet and choosing error. This command appears here.

```
netsh interface ipv6 show ipstats | Select-String errors
```

In the figure that follows, I first show the command to retrieve the IPV6 IP statistics. Next I show the output from the command. Then I filter the output to only errors by using the Select-String cmdlet and lastly, I show the output from the filtered string.

Select Administrator: Windows PowerShell

```
PS C:\> netsh interface ipv6 show  ipstats
MIB-II IP Statistics
------------------------------------------------------
Forwarding is:                      Disabled
Default TTL:                        128
In Receives:                        9508
In Header Errors:                   0
In Address Errors:                  4549
Datagrams Forwarded:                0
In Unknown Protocol:                0
In Discarded:                       3549
In Delivered:                       7926
Out Requests:                       22012
Routing Discards:                   0
Out Discards:                       0
Out No Routes:                      39
Reassembly Timeout:                 60
Reassembly Required:                0
Reassembled Ok:                     0
Reassembly Failures:                0
Fragments Ok:                       0
Fragments Failed:                   0
Fragments Created:                  0


PS C:\> netsh interface ipv6 show ipstats | Select-String errors

In Header Errors:                   0
In Address Errors:                  4550


PS C:\>
```

NetStat

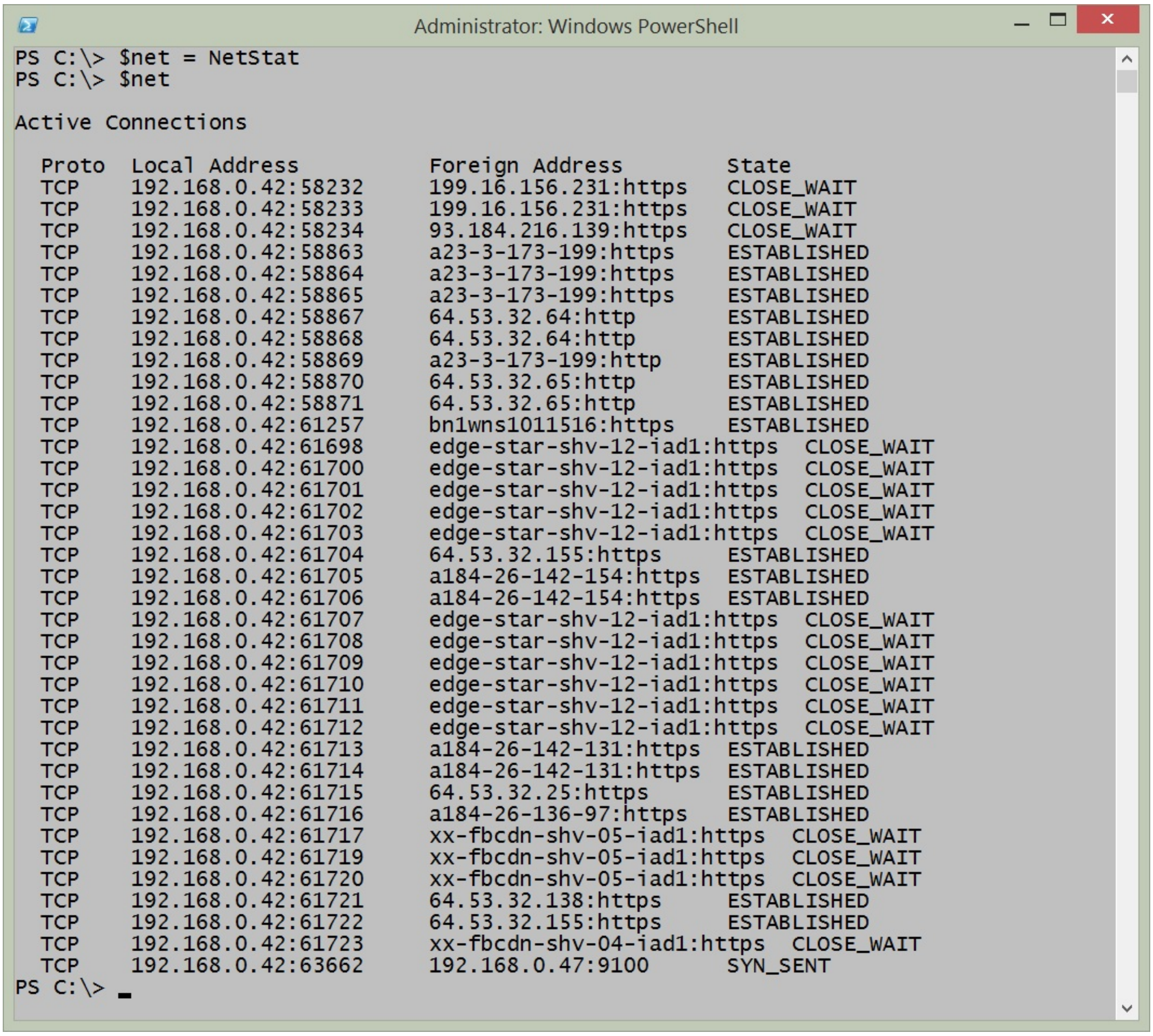
The NetStat command has been around in the Windows world for a long time. It provides a quick snapshot of connections from local ports to remote ports as well as the protocol and the state of those connections. It takes a couple of minutes to run, and as a result it makes sense to store the results of NetStat into a variable. I can then examine the information several times if I wish without having to wait each time to gather the information additional times. Here is an example of running the NetStat command and storing the results from in a variable.

```
$net=NetStat
```

To display the information in an unfiltered fashion, I just type $net at the Windows PowerShell prompt and it displays all of the information that it gathered. Here is an example:

```
$net
```

The command to run NetStat and store the results in a variable as well as to examine the contents of the $net variable appear in the figure that follows.



```
Administrator: Windows PowerShell
PS C:\> $net = NetStat
PS C:\> $net

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    192.168.0.42:58232     199.16.156.231:https   CLOSE_WAIT
  TCP    192.168.0.42:58233     199.16.156.231:https   CLOSE_WAIT
  TCP    192.168.0.42:58234     93.184.216.139:https   CLOSE_WAIT
  TCP    192.168.0.42:58863     a23-3-173-199:https    ESTABLISHED
  TCP    192.168.0.42:58864     a23-3-173-199:https    ESTABLISHED
  TCP    192.168.0.42:58865     a23-3-173-199:https    ESTABLISHED
  TCP    192.168.0.42:58867     64.53.32.64:http       ESTABLISHED
  TCP    192.168.0.42:58868     64.53.32.64:http       ESTABLISHED
  TCP    192.168.0.42:58869     a23-3-173-199:http     ESTABLISHED
  TCP    192.168.0.42:58870     64.53.32.65:http       ESTABLISHED
  TCP    192.168.0.42:58871     64.53.32.65:http       ESTABLISHED
  TCP    192.168.0.42:61257     bn1wns1011516:https    ESTABLISHED
  TCP    192.168.0.42:61698     edge-star-shv-12-iad1:https  CLOSE_WAIT
  TCP    192.168.0.42:61700     edge-star-shv-12-iad1:https  CLOSE_WAIT
  TCP    192.168.0.42:61701     edge-star-shv-12-iad1:https  CLOSE_WAIT
  TCP    192.168.0.42:61702     edge-star-shv-12-iad1:https  CLOSE_WAIT
  TCP    192.168.0.42:61703     edge-star-shv-12-iad1:https  CLOSE_WAIT
  TCP    192.168.0.42:61704     64.53.32.155:https     ESTABLISHED
  TCP    192.168.0.42:61705     a184-26-142-154:https  ESTABLISHED
  TCP    192.168.0.42:61706     a184-26-142-154:https  ESTABLISHED
  TCP    192.168.0.42:61707     edge-star-shv-12-iad1:https  CLOSE_WAIT
  TCP    192.168.0.42:61708     edge-star-shv-12-iad1:https  CLOSE_WAIT
  TCP    192.168.0.42:61709     edge-star-shv-12-iad1:https  CLOSE_WAIT
  TCP    192.168.0.42:61710     edge-star-shv-12-iad1:https  CLOSE_WAIT
  TCP    192.168.0.42:61711     edge-star-shv-12-iad1:https  CLOSE_WAIT
  TCP    192.168.0.42:61712     edge-star-shv-12-iad1:https  CLOSE_WAIT
  TCP    192.168.0.42:61713     a184-26-142-131:https  ESTABLISHED
  TCP    192.168.0.42:61714     a184-26-142-131:https  ESTABLISHED
  TCP    192.168.0.42:61715     64.53.32.25:https      ESTABLISHED
  TCP    192.168.0.42:61716     a184-26-136-97:https   ESTABLISHED
  TCP    192.168.0.42:61717     xx-fbcdn-shv-05-iad1:https  CLOSE_WAIT
  TCP    192.168.0.42:61719     xx-fbcdn-shv-05-iad1:https  CLOSE_WAIT
  TCP    192.168.0.42:61720     xx-fbcdn-shv-05-iad1:https  CLOSE_WAIT
  TCP    192.168.0.42:61721     64.53.32.138:https     ESTABLISHED
  TCP    192.168.0.42:61722     64.53.32.155:https     ESTABLISHED
  TCP    192.168.0.42:61723     xx-fbcdn-shv-04-iad1:https  CLOSE_WAIT
  TCP    192.168.0.42:63662     192.168.0.47:9100      SYN_SENT
PS C:\>
```

The real power, however, comes in using Windows PowerShell to parse the text output to find specific information. For example, the previous output shows multiple connections in various states of connectiveness. I can easily parse the output and find only the connections that are Established. The command I use appears here.

```
$net|select-string "Established"
```

The command and the output from the command appear in the figure that follows.

```
Administrator: Windows PowerShell
PS C:\> $net | select-string "Established"

  TCP    192.168.0.42:58863     a23-3-173-199:https    ESTABLISHED
  TCP    192.168.0.42:58864     a23-3-173-199:https    ESTABLISHED
  TCP    192.168.0.42:58865     a23-3-173-199:https    ESTABLISHED
  TCP    192.168.0.42:58867     64.53.32.64:http       ESTABLISHED
  TCP    192.168.0.42:58868     64.53.32.64:http       ESTABLISHED
  TCP    192.168.0.42:58869     a23-3-173-199:http     ESTABLISHED
  TCP    192.168.0.42:58870     64.53.32.65:http       ESTABLISHED
  TCP    192.168.0.42:58871     64.53.32.65:http       ESTABLISHED
  TCP    192.168.0.42:61257     bn1wns1011516:https    ESTABLISHED
  TCP    192.168.0.42:61704     64.53.32.155:https     ESTABLISHED
  TCP    192.168.0.42:61705     a184-26-142-154:https  ESTABLISHED
  TCP    192.168.0.42:61706     a184-26-142-154:https  ESTABLISHED
  TCP    192.168.0.42:61713     a184-26-142-131:https  ESTABLISHED
  TCP    192.168.0.42:61714     a184-26-142-131:https  ESTABLISHED
  TCP    192.168.0.42:61715     64.53.32.25:https      ESTABLISHED
  TCP    192.168.0.42:61716     a184-26-136-97:https   ESTABLISHED
  TCP    192.168.0.42:61721     64.53.32.138:https     ESTABLISHED
  TCP    192.168.0.42:61722     64.53.32.155:https     ESTABLISHED

PS C:\>
```

Interestingly enough, I can also use NetSh to report on TCP connections. The command appears here:

```
netsh interface ipv4 show tcpconnections
```

The output from the command, as appears in the figure that follows, is a bit different than that received from NetStat.

```
Administrator: Windows PowerShell
PS C:\> netsh interface ipv4 show tcpconnections

MIB-II TCP Connection Entry
Local Address    Local Port      Remote Address  Remote Port     State
-----------------------------------------------------------------------------
        0.0.0.0       135              0.0.0.0           0       Listen
 169.254.140.58       139              0.0.0.0           0       Listen
   192.168.0.42       139              0.0.0.0           0       Listen
        0.0.0.0       554              0.0.0.0           0       Listen
        0.0.0.0      2179              0.0.0.0           0       Listen
      127.0.0.1      2559              0.0.0.0           0       Listen
        0.0.0.0     49152              0.0.0.0           0       Listen
        0.0.0.0     49153              0.0.0.0           0       Listen
        0.0.0.0     49154              0.0.0.0           0       Listen
        0.0.0.0     49155              0.0.0.0           0       Listen
        0.0.0.0     49156              0.0.0.0           0       Listen
        0.0.0.0     49157              0.0.0.0           0       Listen
        0.0.0.0     49160              0.0.0.0           0       Listen
   192.168.0.42     58232       199.16.156.231         443   Close Wait
   192.168.0.42     58233       199.16.156.231         443   Close Wait
   192.168.0.42     58234       93.184.216.139         443   Close Wait
   192.168.0.42     58863          23.3.173.199        443  Established
   192.168.0.42     58864          23.3.173.199        443  Established
   192.168.0.42     58865          23.3.173.199        443  Established
   192.168.0.42     58867           64.53.32.64         80  Established
   192.168.0.42     58868           64.53.32.64         80  Established
   192.168.0.42     58869          23.3.173.199         80  Established
   192.168.0.42     58870           64.53.32.65         80  Established
   192.168.0.42     58871           64.53.32.65         80  Established
   192.168.0.42     61257          157.56.98.92        443  Established
   192.168.0.42     61698          31.13.69.160        443   Close Wait
   192.168.0.42     61700          31.13.69.160        443   Close Wait
   192.168.0.42     61701          31.13.69.160        443   Close Wait
   192.168.0.42     61702          31.13.69.160        443   Close Wait
   192.168.0.42     61703          31.13.69.160        443   Close Wait
```

Performance Counters

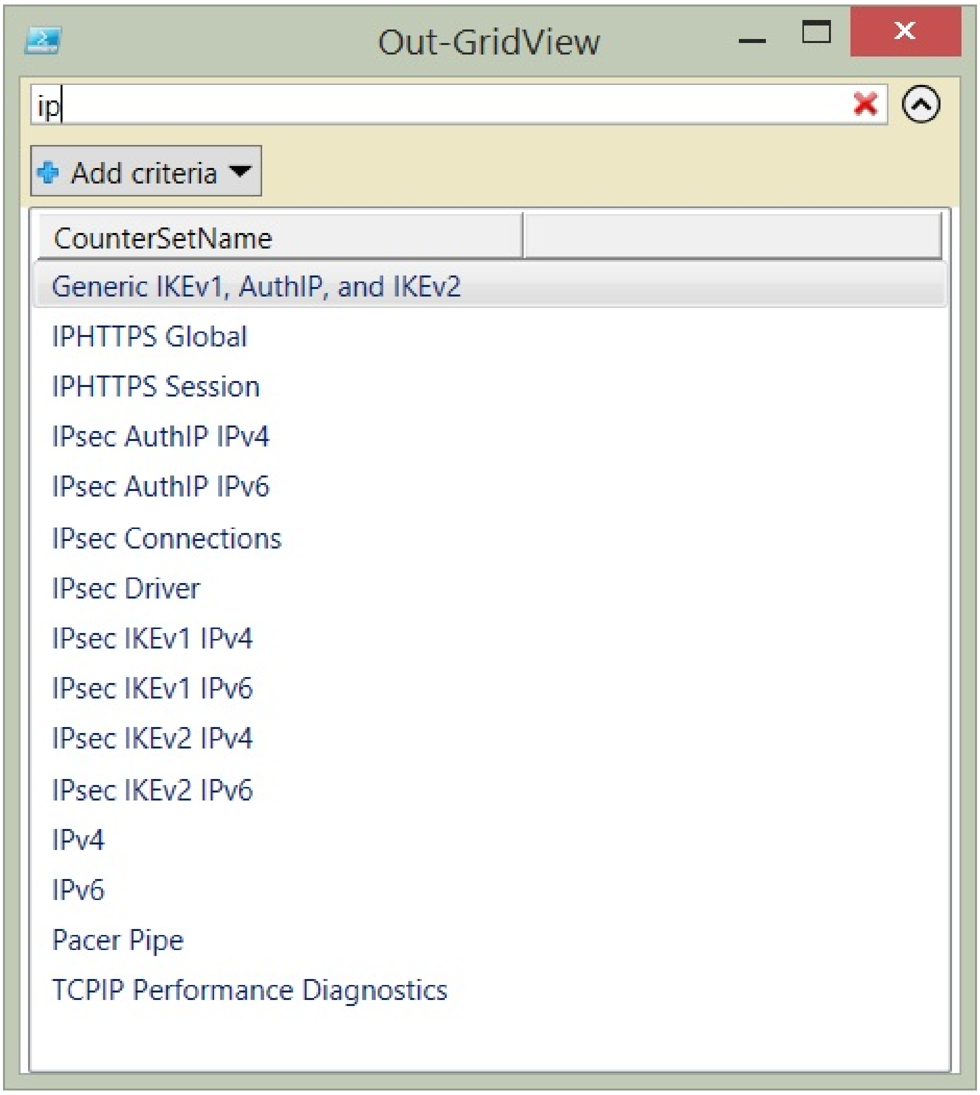
To collect performance counter information, I need to know the performance counter set names so I can easily gather the information. To do this, I use the Get-Counter cmdlet and I choose all of the listsets. I then like to sort on the countersetName property and then select only that property. The following command retrieves the available listsets.

```
Get-Counter -ListSet * |
Sort-Object CounterSetName |
Select-Object CounterSetName
```

If I pipeline the output to the Out-GridView cmdlet, then I can easily filter the list to find the listsets I wish to use. This command appears here.

```
Get-Counter -ListSet * |
Sort-Object CounterSetName |
Select-Object CounterSetName |
Out-GridView
```

The resulting Out-GridView pane makes it easy to filter for different values. For example, the figure that follows filters for IP.



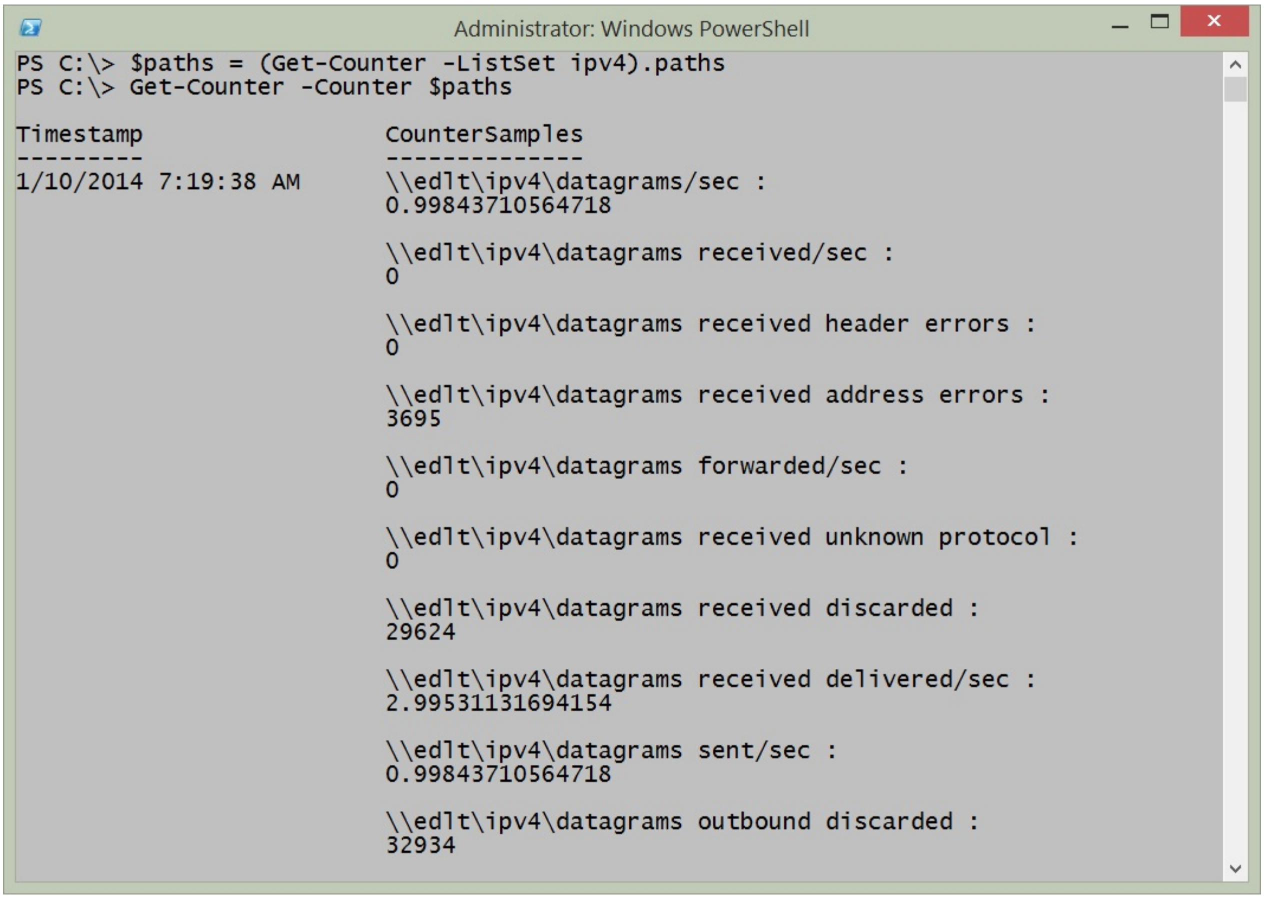
Once I have the countersetname value I wish to query, it is a simple matter of plugging it into the Get-Counter to first obtain the paths. This command appears here.

```
$paths = (Get-Counter -ListSet ipv4).paths
```

Next I use the paths with the Get-Counter cmdlet to retrieve a single instance of the IPv4 performance information. The command appears here.

```
Get-Counter -Counter $paths
```

The commands and the output from the commands appear in the figure that follows.



```
PS C:\> $paths = (Get-Counter -ListSet ipv4).paths
PS C:\> Get-Counter -Counter $paths

Timestamp                 CounterSamples
---------                 --------------
1/10/2014 7:19:38 AM      \\edlt\ipv4\datagrams/sec :
                          0.99843710564718

                          \\edlt\ipv4\datagrams received/sec :
                          0

                          \\edlt\ipv4\datagrams received header errors :
                          0

                          \\edlt\ipv4\datagrams received address errors :
                          3695

                          \\edlt\ipv4\datagrams forwarded/sec :
                          0

                          \\edlt\ipv4\datagrams received unknown protocol :
                          0

                          \\edlt\ipv4\datagrams received discarded :
                          29624

                          \\edlt\ipv4\datagrams received delivered/sec :
                          2.99531131694154

                          \\edlt\ipv4\datagrams sent/sec :
                          0.99843710564718

                          \\edlt\ipv4\datagrams outbound discarded :
                          32934
```

If I want to monitor a counter set for a period of time, I use the -SampleInterval property and the -MaxSamples parameter. In this way I can specify how long I want the counter collection to run. An example of this technique appears here.

```
Get-Counter -Counter $paths -SampleInterval 60 -MaxSamples 60
```

If I want to monitor continuously, until I type Ctrl-C and break the command, I use the -Continuous parameter and the -SampleInterval parameter. An example of this command appears here.

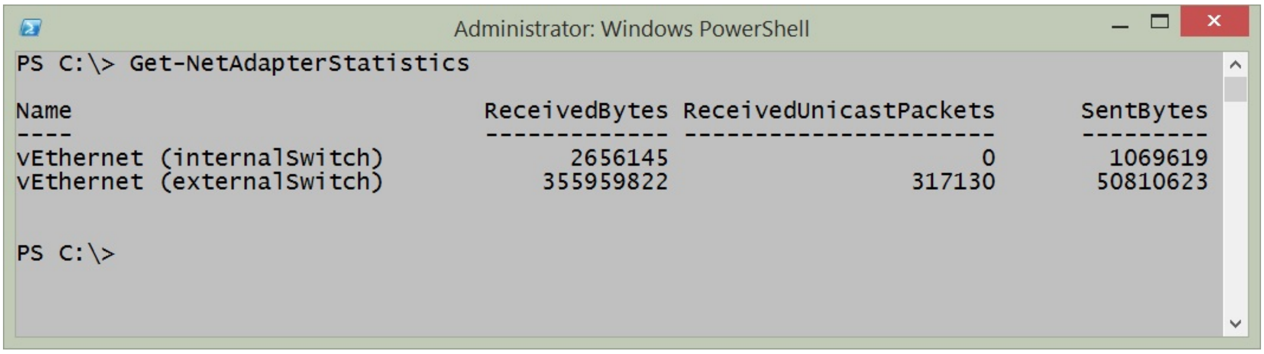
```
Get-Counter -Counter $paths -SampleInterval 30 -Continuous
```

Using Get-NetAdapterStatistics function

The easiest way to gather network adapter statistics is to use the Get-NetAdapterStatistics function from the NetAdapter module. It provides a quick overview of the sent and received packets. An example of the command appears here.

```
Get-NetAdapterStatistics
```

The command and a sample output appear in the figure that follows.



```
Administrator: Windows PowerShell
PS C:\> Get-NetAdapterStatistics

Name                          ReceivedBytes ReceivedUnicastPackets       SentBytes
----                          ------------- ----------------------       ---------
vEthernet (internalSwitch)          2656145                      0         1069619
vEthernet (externalSwitch)        355959822                 317130        50810623


PS C:\>
```

If I want to work with a specific network adapter I can use the name of the adapter, or for more flexibility I can pipeline the results from the Get-Netadapter function. This technique appears here.

```
Get-NetAdapter -ifIndex 12| Get-NetAdapterStatistics
```

The Get-NetAdapterStatistics function returns more than just bytes sent and received. To find the additional information I like to pipeline the results to the Format-List cmdlet. An example of this technique appears here.

```
Get-NetAdapter -ifIndex 12| Get-NetAdapterStatistics | format-list\*
```

The command, and the output associated with the command appear in the figure that follows.

Administrator: Windows PowerShell

```
PS C:\> Get-NetAdapter -ifIndex 12 | Get-NetAdapterStatistics | format-list *


ifAlias                    : vEthernet (externalSwitch)
InterfaceAlias             : vEthernet (externalSwitch)
ifDesc                     : Hyper-V Virtual Ethernet Adapter #2
Caption                    : MSFT_NetAdapterStatisticsSettingData 'Hyper-V Virtual
                             Ethernet Adapter #2'
Description                : Hyper-V Virtual Ethernet Adapter #2
ElementName                : Hyper-V Virtual Ethernet Adapter #2
InstanceID                 : {1CADC0D8-CD3D-43AF-A44F-62FD4D55CEC5}
InterfaceDescription       : Hyper-V Virtual Ethernet Adapter #2
Name                       : vEthernet (externalSwitch)
Source                     : 2
SystemName                 : edlt.iammred.net
OutboundDiscardedPackets   : 0
OutboundPacketErrors       : 0
RdmaStatistics             :
ReceivedBroadcastBytes     : 0
ReceivedBroadcastPackets   : 57099
ReceivedBytes              : 356004173
ReceivedDiscardedPackets   : 0
ReceivedMulticastBytes     : 0
ReceivedMulticastPackets   : 37266
ReceivedPacketErrors       : 0
ReceivedUnicastBytes       : 0
ReceivedUnicastPackets     : 317161
RscStatistics              :
SentBroadcastBytes         : 0
SentBroadcastPackets       : 36262
SentBytes                  : 50819157
SentMulticastBytes         : 0
SentMulticastPackets       : 4204
SentUnicastBytes           : 0
SentUnicastPackets         : 273247
SupportedStatistics        : 975
PSComputerName             :
CimClass                   : ROOT/StandardCimv2:MSFT_NetAdapterStatisticsSettingData
CimInstanceProperties      : {Caption, Description, ElementName, InstanceID...}
CimSystemProperties        : Microsoft.Management.Infrastructure.CimSystemProperties


PS C:\>
```

Resources

Books from Microsoft Press

Windows PowerShell 3.0 First Steps - A Windows PowerShell primer providing an overview of the major Windows PowerShell components.

Windows PowerShell 3.0 Step by Step - A Windows PowerShell step by step learning guide, complete with lab exercises, review questions, and answers. This book contains hundreds of Windows PowerShell scripts.

Windows PowerShell Best Practices - The Windows PowerShell best practices guide, containing real world tips, gotchas, and techniques from hundreds of field personnel. The contributors include Microsoft Windows PowerShell developers, Microsoft Windows PowerShell MVP's, Enterprise network administrators, and top Dev-ops.

Web Resources

The Microsoft Script Center - dedicated to system administrator scripters the world over.

The Scripting Guys Forum - community forum for asking scripting questions.

The Script Center Script Repository - the largest collection of admin scripts on the internet.

The Hey Scripting Guy Blog - thousands of blog articles about scripting. Updated twice a day, 365 days a year, it is the #1 blog on MSDN and on TechNet.

The Script Center Community Page - insight into Windows PowerShell community activities, especially activities where the Microsoft Scripting Guy and the Scripting Wife will appear.

The Script Center Learn PowerShell Page - central hub for learning about Windows PowerShell.

NetAdapter module documentation - official Microsoft documentation for the NetAdapter module from TechNet.

PoshCode - Windows PowerShell community driven script repository.

PowerShell.Org - Windows PowerShell community site containing blogs, forums, user group information and a script repository.

PowerShellGrroup.org - listing of Windows PowerShell community user group meetings.

PowerShellSaturday.Org - listing of Windows PowerShell Saturday community events.